跟老齐学

Python Django实战

第2版

齐伟 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

VISIT…

内 容 简 介

本书是以 Python 为基础进行 Web 应用开发的进阶读物。书中以一个实例项目为主线，使用 Django 2 开发框架，在实践中边学边做，理论联系实际。每节都配有思维导图，使读者对项目需求一目了然；每章都有知识点和文档导读，引导读者“知其所以然”。相信认真阅读本书的读者，不仅能够得到“鱼”，更能得到“渔”，从而具备独立开发项目的能力。

本书适合已经具有 Python 基础技能、进行 Web 应用开发的读者阅读。

如有此意愿，但尚缺乏 Python 基础技能者，推荐阅读《跟老齐学 Python：轻松入门》。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

跟老齐学 Python：Django 实战 / 齐伟编著. —2 版. —北京：电子工业出版社，2019.1

ISBN 978-7-121-35503-5

Ⅰ. ①跟… Ⅱ. ①齐… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2018）第 258023 号

策划编辑：高洪霞
责任编辑：牛　勇　　特约编辑：赵树刚
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×1092　1/16　　印张：17.5　　字数：448 千字
版　　次：2017 年 10 月第 1 版
　　2019 年 1 月第 2 版
印　　次：2019 年 1 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

序

按照 Django 的版本发布计划，现在已经是 Django 2 的时代了，所以，本书也要跟进。在本书第 2 版的代码中，使用了 Django 2，源码依然放在：https://github.com/qiwsir/DjangoPracticeProject。

本书的第 2 版相对第 1 版，不仅是 Django 版本的迭代，对部分内容也进行了修改，采用了读者对第 1 版提出的各种意见和建议，在此对众多热心读者表示感谢。

Web 开发所需要的技术比较庞杂，那些“从前到后”都能做的开发者被称为“全栈工程师”。但是，如果本书宣称“全栈”，肯定是骗人的——请读者注意，凡是宣称“全栈”的书或课程，都要小心。要达到“全栈”的目的，不仅需要读书，更需要项目的积累。

为了协助读者了解 Web 开发的各种相关技术，我特意配合本书开发了一个小程序（名为：跟老齐学）。希望能够通过小程序，实现“把书读厚”的目的。至于“读薄”，抑或“放弃”，则应视读者个人的修为而定了。

扫描二维码，关注小程序（小程序名称：跟老齐学）。

除了小程序之外，QQ 群（新的群号：746028518）依然是读者之间的交流园地；网站（itdiffer.com）也会一如既往地提供相关阅读资料。

非常感谢编辑老师为本书所付出的辛苦工作，以及刘燕清先生在百忙之中帮助我调试代码，在此一并表示感谢！

齐 伟

2018 年 7 月

前言

是的，你应该开始阅读本书了。

如果读者已经通过《跟老齐学 Python：轻松入门》这本书完成了 Python 入门，那么接下来就要用 Python 做一些实际项目了，做什么呢？可以做的东西真不少，比如数据统计、爬虫、机器学习、人工智能等，当然还有不可或缺的 Web 开发。

本书就是要带领读者：

- 学习 Django（特别声明，本书中的项目选用 Django 2）；
- 完成一个项目；
- 边做项目边学知识。

因此，本书按照完成一个项目从易到难的顺序，以项目为主线逐渐展开对 Django 的学习。Django 是一种在 Web 开发中应用非常广泛的开发框架，它能够让程序员非常快捷地建设一个网站，并且支持网站开发中各种复杂的业务。如果需要快速建设一个网站，并且希望后期还能通过迭代开发实现网站功能和性能的提升，那么 Django 是一个不错的选择。

Django 的官方网站中有不少内容完备的文档，当然，这种完备不是针对初学者而言的。对于初学者来说，也许官方文档太简要了，因此本书的作用就是将初学者领入 Django 开发的大门，并引导读者熟悉使用 Django 官方文档。当读者阅读并练习完本书的项目后，就能够使用官方文档进行开发了，本书的目的也就达到了。

本书从始至终贯穿着一个项目实践，将开发实战中常用的 Django 知识串联起来，因此它不像课本那样按部就班地把每个知识点都罗列出来，而是根据项目需要，将技能融汇在实战之中。此外，如果仅从开发实践的角度学习 Django，难免会出现就问题而解决问题的现象，缺少系统和深入的学习。为此，书中的每个小节都配有“知识点”，用以扩展或者系统化有关知识，特别是在“文档导读”中会向读者提供有关网络资料，试图让读者知其然并知其所以然。

在项目进行中，读者可能会遇到暂时无法理解的部分，不要纠结于一时之惑，而要跟着项目继续进行下去，或许到后面就能理解前面的内容了——“发展是硬道理”。

如果正在阅读本书的你是 Web 开发的初学者，此前没有做过任何网站开发，那么笔者特别建议遵循如下学习方法。

- 严格按照本书要求，将所有代码都认真敲过。不要复制代码（本书提供的源代码，仅仅是一个参考），要自己一个字母一个字母地敲进去。
- 将自己的调试结果与本书所示结果进行对照，看看是否实现了预期的功能。
- 在调试不能通过时，首先要认真检查自己的代码，包括但不限于拼写问题、缩进问题等，若还不能解决，可通过 Google 搜索该异常（错误），请相信，你的错误别人也同样遇到过。
- 本书所支持的网站 www.itdiffer.com 和 QQ 群（26913719）也是读者学习的助手。
- 本书提示读者阅读的官方文档，一定要认真阅读。
- 本书实际上是一个 Django 项目，项目中包含了多个应用。如果读者按照本书的要求，第一遍就实现了应用的功能，那只能说明本书描述无误且你的阅读能力尚可，并不代表你已经掌握了什么。只有当你把刚刚调试通过的应用删除，然后完全靠自己的记忆和理解，同样实现了该应用的功能，才能说明你掌握了相关知识技能。再次强调，要理解相关内容，就必须多重复几遍。

因为本书是以“项目为主线”的，所以不会面面俱到地将 Django 的所有内容介绍一遍，而是根据项目的功能需要，有选择地使用和介绍有关内容。对于具体功能的实现，书中所给出的代码也非唯一写法，读者可以使用其他更好的方式实现。

在实际的 Web 开发项目中，除使用 Django 框架外，还会用到 HTML、CSS、JavaScript 等知识，从而做出一个看起来有点设计感的页面。虽然说“人不可貌相”，但使用 Django 做出一个漂亮的前端，还是能够让人感觉很舒服的。因此，书中也涉及一些前端内容，读者在阅读时，如果缺少相关知识，可以实时补充。当然，前端知识并不是学习本书所必备的，因为凭借读者的聪明才智和无所不包的网络，掌握应付本书所需要的前端知识是非常容易的。

读者可以在 GitHub 上（https://github.com/qiwsir/DjangoPracticeProject）获得本书的所有代码，这些代码是最终结果，希望不会给读者的学习带来不便。

感谢为本书的面市提供帮助的编辑们，感谢我的妻子在本书编写过程中提供的支持。

齐　伟

2018 年 8 月

目录

第 1 章　简单的博客系统 1

1.1　Django 起步 1

1.1.1　Django 简介 1

1.1.2　安装 Django 3

1.1.3　创建项目 4

1.1.4　创建应用 6

1.1.5　网站配置 12

1.1.6　知识点 13

1.2　编写博客的数据模型类 13

1.2.1　数据模型类 14

1.2.2　发布博客文章 18

1.2.3　知识点 22

1.3　显示博客信息 24

1.3.1　显示文章标题 25

1.3.2　查看文章内容 30

1.3.3　知识点 33

第 2 章　用户管理 37

2.1　自定义模板和静态文件位置 37

2.1.1　自定义模板位置 38

2.1.2　自定义静态文件位置 38

2.1.3　通用静态文件和基础模板 39

2.1.4　重置管理后台模板 43

2.1.5　知识点 45

2.2　用户登录 46

2.2.1 创建应用 47
2.2.2 理解表单类 48
2.2.3 登录的视图函数 49
2.2.4 登录的前端界面 52
2.2.5 知识点 56
2.3 用内置方法实现登录和退出 58
2.3.1 内置的登录方法 58
2.3.2 判断用户是否登录 60
2.3.3 内置的退出方法 61
2.3.4 知识点 62
2.4 用户注册 64
2.4.1 简单注册 64
2.4.2 增加注册内容 67
2.4.3 管理新增的注册内容 71
2.4.4 知识点 72
2.5 关于密码的操作 74
2.5.1 修改密码 74
2.5.2 重置密码 79
2.5.3 知识点 85
2.6 维护个人信息 87
2.6.1 个人信息的数据模型类和表单类 87
2.6.2 展示个人信息 88
2.6.3 编辑个人信息 91
2.6.4 上传和裁剪头像图片 95
2.6.5 优化头像上传功能 103
2.6.6 对个人信息进行管理 106
2.6.7 知识点 107
第 3 章 文章管理和展示 109
3.1 管理文章栏目 109
3.1.1 设置栏目 109
3.1.2 编辑栏目 117
3.1.3 删除栏目 119
3.1.4 知识点 121

3.2 发布和显示文章 124
3.2.1 简单的文章发布 125
3.2.2 使用 Markdown 130
3.2.3 文章标题列表 132
3.2.4 知识点 140
3.3 删除和修改文章 141
3.3.1 删除 141
3.3.2 修改 143
3.3.3 设置分页功能 147
3.3.4 知识点 149
3.4 文章展示 151
3.4.1 新写文章标题列表 151
3.4.2 重新编写“查看文章”功能 155
3.4.3 知识点 156
第 4 章 拓展网站功能 158
4.1 梳理已有功能 158
4.1.1 修改导航栏 158
4.1.2 修改登录和注册后的跳转 159
4.1.3 知识点 161
4.2 查看作者全部文章 162
4.2.1 查看某作者的文章列表 162
4.2.2 知识点 166
4.3 为文章点赞 167
4.3.1 修改数据模型类 167
4.3.2 编写视图函数 168
4.3.3 修改模板文件 169
4.3.4 知识点 173
4.4 文章的阅读次数 174
4.4.1 安装 Redis 175
4.4.2 在 Python 中使用 Redis 176
4.4.3 记录阅读次数 177
4.4.4 显示最“热”文章 178
4.4.5 知识点 180

4.5 文章的评论功能 181
4.5.1 数据模型类和表单类 181
4.5.2 实现评论功能 183
4.5.3 知识点 185
4.6 多样化显示 186
4.6.1 统计文章总数 187
4.6.2 最新发布的文章 190
4.6.3 评论最多的文章 191
4.6.4 自定义模板选择器 194
4.6.5 知识点 195
4.7 管理和应用文章标签 196
4.7.1 管理文章标签 197
4.7.2 发布文章时选择标签 202
4.7.3 在文章中显示文章标签 204
4.7.4 推荐相似文章 205
4.7.5 知识点 207
第 5 章 收集和展示图片 208
5.1 收集网络图片 208
5.1.1 创建图片相关类 209
5.1.2 收集和管理图片 212
5.1.3 完善图片管理功能 217
5.1.4 知识点 220
5.2 展示图片 221
5.2.1 瀑布流方式展示图片 222
5.2.2 查看图片的详细信息 225
5.2.3 知识点 226
第 6 章 中场休整 228
第 7 章 创建在线学习应用 229
7.1 基于类的视图 229
7.1.1 最简类视图 230
7.1.2 读取数据 232
7.1.3 初步了解 Mixin 235
7.1.4 知识点 236

7.2 管理课程标题 237
7.2.1 判断用户是否登录 237
7.2.2 创建课程 240
7.2.3 删除课程 242
7.2.4 知识点 247
7.3 发布和学习课程内容 249
7.3.1 课程内容的数据模型 250
7.3.2 课程内容的表单类 253
7.3.3 课程内容的视图 253
7.3.4 查看课程内容 256
7.3.5 注册学习课程 260
7.3.6 知识点 264
第 8 章 结束和开始 266

第1章

简单的博客系统

从现在开始，请读者随我一起做一个项目，这个项目的名称叫作“多用户内容发布系统”。因为很多读者是来学习的，所以还要本着“循序渐进”“深入浅出”的原则来做这个项目——这当然不是工程项目开发的原则，这是教学的原则。

本项目从博客开始。

博客，现在已经有人把它看作历史文物了，因为互联网的迅猛发展，10 年前的事物就算古董了。尽管如此，博客依然可以作为一个学习的样本。本章将通过一个单用户的博客系统的开发，初步说明利用 Django 开发网站的基本步骤和网站的基本组成。当然，这仅仅是起步，并不意味着学完本章就理解 Django 了。“不积跬步，无以至千里”，最后的复杂系统也是由一个个简单的小功能堆砌起来的。

1.1 Django 起步

Django 这个词对一些人来说或许并不陌生，有一部荣获第 85 届奥斯卡金像奖的影片叫作 *Django Unchained*，中文名被翻译为《被解救的姜戈》，这或许是 Django 首度被翻译为中文。但是，作为网站开发框架的 Django 与这部电影没有任何关系。

Django 的起步内容如图 1-1 所示。

1.1.1 Django 简介

一般认为 Django 开发框架（简称 Django）诞生的时间是 2003 年的金秋时节。此时千年古城苏州正桂花飘香，而大洋彼岸的美国有两位程序员在使用后来被冠名为 Django 的框架做网站开发，这是全世界首次使用。让我们感谢这两位程序员，他们是 Adrian Holovaty 和 Simon Willison。2008 年虽然遇到了金融危机，但依然挡不住技术的进步，Django 有了自己的 DSF（Django Software Foundation），后来 Django 的发展就由这个基金会来主导了。

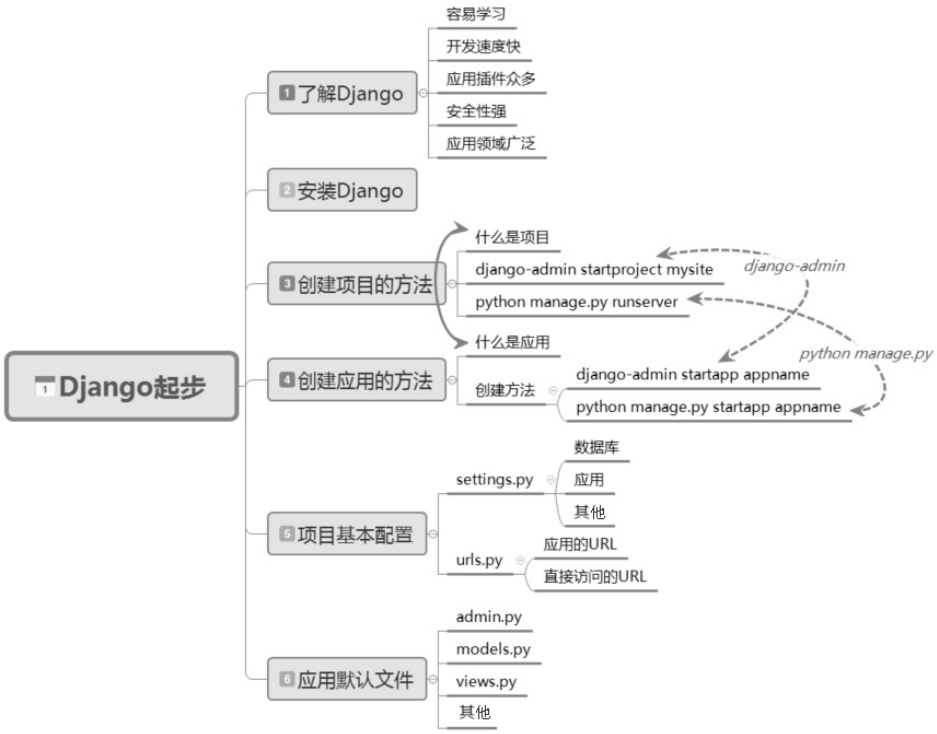

图 1-1　知识技能导图

对于 Django 的评价，借用李清照的《鹧鸪天 • 桂花》来表达，我认为是非常恰当的。

> 暗淡轻黄体性柔。情疏迹远只香留。何须浅碧深红色，自是花中第一流。
>
> 梅定妒，菊应羞。画栏开处冠中秋。骚人可煞无情思，何事当年不见收。

“自是花中第一流”，不仅仅是现在，未来依然蓬勃发展。在 Django 官方网站披露的信息中，可以看到其发展蓝图，如图 1-2 所示。

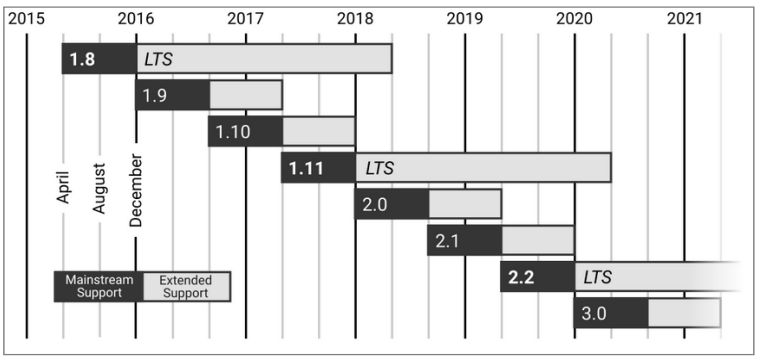

图 1-2　Django 各版本计划

关于 Django 的特点，用官方网站（https://www.djangoproject.com）上的大标题就可以概括了：“Django makes it easier to build better Web apps more quickly and with less code”。展开来说，就是如下几点。

- 容易上手，开发速度快。
- 囊括了网站开发中的用户管理、内容管理、网站地图、RSS 等常用的众多插件。
- 安全性强，比如 Django 默认解决了 SQL 注入、跨站攻击等问题。
- 应用广泛，类型多样化。使用 Django 开发的网站包括各类在线服务网站、社会组织和政府机构网站等，其类型包括但不限于管理系统、社交网站、计算平台等。

就开发网站的框架而言，除 Django 外，在 Python 领域还有 Tornado、Flask 等，它们各有各的特点，但 Django 的应用范围最广。

接下来就开始 Django 之旅——虽有解救姜戈的惊心动魄，但无生命之忧。

1.1.2　安装 Django

在一般情况下，读者所用的计算机操作系统上没有 Django，需要自己安装。

Django 是以 Python 为语言环境的，所以要先保证计算机上已经安装了 Python。读者如果对 Python 知之甚少，可以参阅《跟老齐学 Python：轻松入门》，这本书介绍了 Python 的基础知识。

Django 适用于 Python 3 和 Python 2.7 两种版本，如果你是一个新秀，并不是为了承接基于 Python 2.7 的旧项目开发，可以直接用 Python 3 进行 Django 的学习和开发。

从官方网站可以获悉，Django 项目团队正在不断发展 Django 的版本（见图 1-2），本书第 1 版在 2017 年刚刚发行的时候，使用的是 Django 1.10。2018 年，Django 已经演进到 Django 2.x 了。Django 2.x 相对于 Django 1.x 有部分变化，因此本书再次修订，转而采用 Django 2.x。读者在阅读的时候，可能 Django 的版本已经高于本书所使用的版本了，但是，影响不大，因为都是 Django 2.x 的范畴，小版本号的修改，仅意味着些许的优化。使用本书进行学习的时候，可以安装使用最新版的 Django，不用在意小版本号的更改。

安装 Django 的最简单方法是输入以下命令（关于 pip，请参阅《跟老齐学 Python：轻松入门》有关章节说明）：

```
$ pip3 install Django
```

安装好 Django 之后，可以先进入 Python 交互模式中，查看一下所安装的 Django 版本。

```
$ python3
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import django
>>> print(django.get_version())    #打印出当前 Django 的版本号
2.1.4
```

上面的安装方法是一种简单而且常用的方法，除这种方法外，还可以下载 Django 源码进行安装：

```
git clone https://github.com/django/django.git
```

如果系统中没有安装 Git，会提示用户安装。

这时，会在当前目录中看到一个名称为“django”的目录，里面是最新版本的 Django。

随后在当前目录中进行如下操作：

```
$ pip3 install -e ./django
```

安装完毕，系统会提示“Successfully installed Django”。

通过这种方法得到的必然是 Django 的最新版本，而且是正在开发中的最新版，尚未正式发布——虽然如此，也是能够正常使用的，Django 的开发者不会把不能使用的代码发布出来。喜欢尝试的读者可以用上面这种方式安装。

安装成功之后，在 Python 交互模式中可以查看当前 Django 版本，显示结果中包含“dev”字样，说明我们使用的是开发版。

在一般情况下，推荐使用 pip install Django 的方法来安装，其原因就是笔者提倡的保守主义。

基础设施已经建好，下面就要创建项目了。

1.1.3 创建项目

Django 中的“项目”（project）可以看作一个专有名词，因为后面还有一个与之有关的名词“应用”（application）。

所谓“项目”，可以理解为一个网站。

先规划好将项目创建在什么地方，比如，笔者放在如下的目录中：

```
Chees-MBP:mysite2 qiwsir$ pwd
/Users/qiwsir/Documents/Codes/DjangoPracticeProject/mysite2
```

在刚刚创建的 mysite2 目录中，创建一个 Django 项目：

```
Chees-MBP:mysite2 qiwsir$ django-admin startproject mysite
```

看一下这个目录，多了一个 mysite 子目录，mysite 就是这个项目的名称：

```
Chees-MBP:mysite2 qiwsir$ ls
mysite
```

mysite 子目录里面的内容，就是我们创建的项目内容，如图 1-3 所示。

图 1-3　mysite 的目录结构

这是创建项目的一种方法。

可以把刚才建立的项目删除，即删除./mysite2/mysite/目录，然后尝试另外一种创建项目的方法，请读者注意观察两种方法的命令形式和结果。第二种创建项目的方法如下：

```
Chees-MBP:mysite2 qiwsir$ django-admin startproject mysite .
```

这次在项目名称 mysite 后面有一个空格，然后是一个句点（英文半角句号），如此创建项目，旨在将项目内容放到当前目录（./mysite2/）中。创建项目之后，仔细观察目录结构，如图 1-4 所示，与图 1-3 所示的目录结构进行对比，从而找出两种方式的区别。

图 1-4　mysite2 中的项目目录结构

在所创建的项目目录中，有一个名称很特殊的文件__init__.py，熟悉 Python 的读者一定知道，其所在目录因它的存在而变成了一个 Python 包（package）（关于包、模块的详细内容请参阅《跟老齐学 Python：轻松入门》）。

此外，与 manage.py 在同一个级别上有一个目录 mysite。如果读者在创建项目时用其他的名称代替 mysite，那么就会出现以该名称命名的目录。

至此，已经建立了一个项目，也就意味着已经有一个网站的基本框架了，虽然现在还不能访问什么。

准备就绪，执行下述操作：

```
$ python3 manage.py runserver
```

在本书中，为了明确说明目录或者文件的位置，以“./”表示项目根目录，如图 1-4 所示的目录结构，manage.py 文件在项目中的位置就是“./manage.py”；在子目录 mysite 中看到的 urls.py 文件，则用“./mysite/urls.py”路径表示。

在上述指令中使用了“python3”，是因为笔者的开发环境配置原因，读者如果在本地的开发环境中以“python”指向了 Python3，也可以使用“python manage.py runserver”指令。

在执行上述指令后，如果一切正常，最终会看到如下的提示信息：

```
$ python3 manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 14 unapplied migration(s). Your project may not work properly until you apply
the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

June 11, 2018 - 03:25:15
Django version 2.1.4, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

由提示信息可知：通过 http://127.0.0.1:8000/访问所创建的项目；按 Ctrl+C 组合键结束当前服务。

打开浏览器，在地址栏中输入 http://127.0.0.1:8000 或者 http://localhost:8000，就会看到如图 1-5 所示的结果。

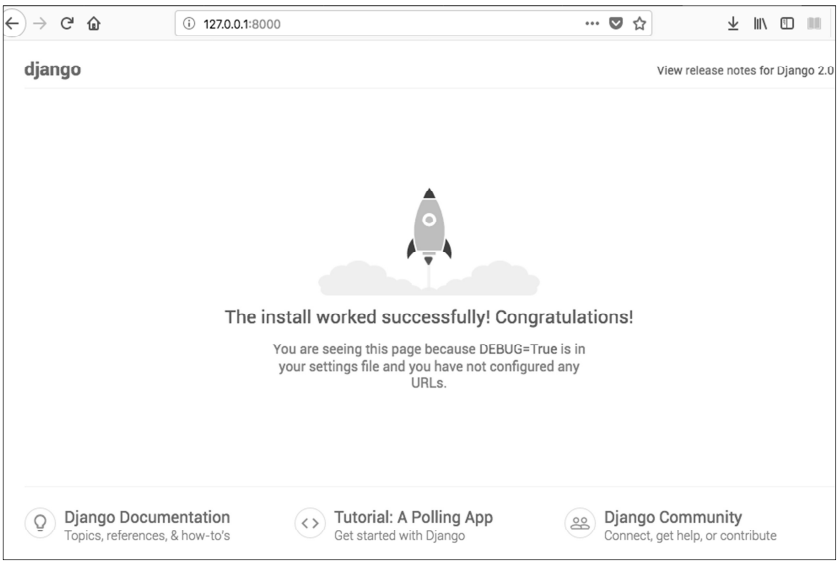

图 1-5　项目默认首页

祝贺！祝贺！热烈祝贺！

已经看到了第一个网页，虽然它很简陋，但未来它将被不断优化，成为一个优秀的网站，读者会用它改变世界——程序员，天生就被赋予了改变世界的使命。某天某布斯找到你，“我现在只差一个程序员了”。为了改变世界，继续学习吧！

网站的成长方式就是不断地增加功能。在 Django 中，人们将完成某个或者某几个功能的集合称为一个“应用”，所以一个功能比较多的网站常常是由多个“应用”组成的，后面我们可以把注意力集中在一个个“应用”上。

1.1.4　创建应用

项目已经创建好，网站也有了，接下来要实现网站的具体功能。在 Django 中，把这些具体的功能称为“应用”（application）。

进入刚才创建的项目目录中，即 manage.py 文件所在的目录，然后按照下面的方式操作：

```
$ ls
db.sqlite3   manage.py    mysite
$ python3 manage.py startapp blog    #①
$ ls
blog     db.sqlite3   manage.py    mysite
```

从上述操作中可以看出，执行了语句①之后，在目录中多了一个名为 blog 的目录。如果观察这时候的 mysite2 目录结构，会看到 blog 里面已经有默认的文件和目录了，如图 1-6 所示。

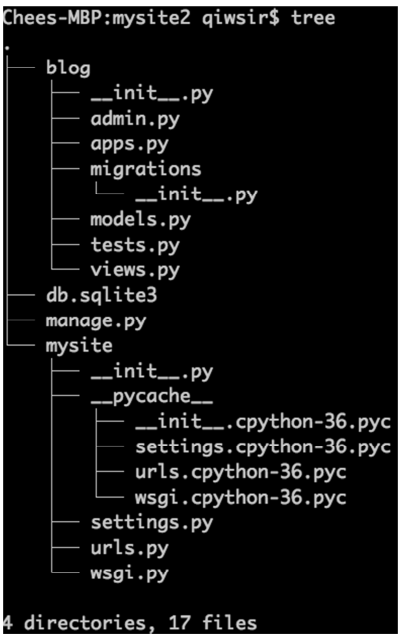

图 1-6　含有 blog 应用的目录结构

blog 就是项目 mysite 中的一个应用。在新的应用创建后，Django 会自动在这个应用中增加一些文件。

请认真观察图 1-6 所示的项目结构，其中不仅有新建的应用 blog，还有 mysite 目录和 manage.py 等文件。目前这已经是一个相对完善的网站项目结构了，下面依次对各个部分进行简要说明。

1．manage.py

在了解 manage.py 之前，先讲解创建项目时用到的 django-admin.py，它是 Django 的任务管理命令行工具。可以通过下面的方式查看 django-admin 命令的帮助信息和所支持的命令行参数：

```
$ django-admin
Type 'django-admin help <subcommand>' for help on a specific subcommand.
Available subcommands:
[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
```

```
    runserver
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver
```

请读者耐心地把这些命令行参数都看一下，留个印象。

看到 startapp 和 startproject 了吧！在创建项目的命令中，我们使用了 startproject。这里还有一个 startapp，这个参数在刚刚创建的操作①中出现了，但不是用在 django-admin 后面，而是用于 python manage.py 的参数，那么这里的 startapp 是什么意思呢？

```
$ django-admin help startapp
usage: django-admin startapp [-h] [--template TEMPLATE]
                             [--extension EXTENSIONS] [--name FILES]
                             [--version] [-v {0,1,2,3}] [--settings SETTINGS]
                             [--pythonpath PYTHONPATH] [--traceback]
                             [--no-color] [--force-color]
                             name [directory]

Creates a Django app directory structure for the given app name in the current
directory or optionally in the given directory.

positional arguments:
  name                  Name of the application or project.
  directory             Optional destination directory

optional arguments:
  -h, --help            show this help message and exit
  --template TEMPLATE   The path or URL to load the template from.
  --extension EXTENSIONS, -e EXTENSIONS
                        The file extension(s) to render (default: "py").
                        Separate multiple extensions with commas, or use -e
                        multiple times.
  --name FILES, -n FILES
                        The file name(s) to render. Separate multiple file
                        names with commas, or use -n multiple times.
  --version             show program's version number and exit
  -v {0,1,2,3}, --verbosity {0,1,2,3}
                        Verbosity level; 0=minimal output, 1=normal output,
                        2=verbose output, 3=very verbose output
```

```
  --settings SETTINGS   The Python path to a settings module, e.g.
                      "myproject.settings.main". If this isn't provided, the
                      DJANGO_SETTINGS_MODULE environment variable will be
                      used.
  --pythonpath PYTHONPATH
                      A directory to add to the Python path, e.g.
                      "/home/djangoprojects/myproject".
  --traceback          Raise on CommandError exceptions
  --no-color           Don't colorize the command output.
  --force-color        Force colorization of the command output.
```

用 django-admin help 来查看一下，发现 startapp 这个参数也是用于创建应用的，与 python3 manage.py startapp application_name 中 startapp 的功能一样。

仔细地阅读帮助信息，然后把刚刚建立的项目中的 blog 应用删除，再用 django-admin startapp blog 试一下：

```
Chees-MBP:mysite2 qiwsir$ ls
blog     db.sqlite3   manage.py    mysite
Chees-MBP:mysite2 qiwsir$ rm -rf ./blog
Chees-MBP:mysite2 qiwsir$ ls
db.sqlite3   manage.py    mysite
Chees-MBP:mysite2 qiwsir$ django-admin startapp blog    #②
Chees-MBP:mysite2 qiwsir$ ls
blog     db.sqlite3   manage.py    mysite
```

语句②和语句①的形式不同，但效果一样，也建立了应用 blog，如果用 tree 命令查看目录结构，会发现与前面的目录结构一模一样。

既然 django-admin startapp blog 和 python manage.py startapp blog 是殊途同归的，那么 manage.py 是什么呢？

在创建一个 Django 项目后，manage.py 在项目的根目录中被自动生成，它是对 django-admin.py 的简单封装，同样能够实现命令行操作：

```
Chees-MBP:mysite2 qiwsir$ python3 manage.py
Type 'manage.py help <subcommand>' for help on a specific subcommand.
Available subcommands:
[auth]
    changepassword
    createsuperuser
[contenttypes]
    remove_stale_contenttypes
[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
```

```
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver
[sessions]
    clearsessions
[staticfiles]
    collectstatic
    findstatic
    runserver
```

与 django-admin 进行对比，两者内容大体相同，同时 manage.py 还有自己的特点。

django-admin 命令对应着 django-admin.py 文件，它在 Django 安装后保存在 Django 安装目录的/bin 下面。

而 mange.py 只在建立了一个项目之后，才存在于项目的根目录中。

关于这两种命令行方式的更多辨析，读者可以阅读官方提供的内容：https://docs.djangoproject.com/en/2.1/ref/django-admin。注意观察这个地址，其中“2.1”表示的是 Django 版本号。如果读者使用的版本与本书有异，一般来讲修改这个版本号即可查看相应内容。

最后提示读者，对于项目根目录中的 manage.py 文件，不要修改，也不要删除，后面我们会经常使用它。

2．mysite

mysite 是所建项目的管理功能目录（不是项目所在的目录 mysite2），这个目录的名称因用户所创建的项目名称的不同而异，虽然可以在这个目录中输入应用部分的内容，但在类似本书的多应用项目中一般不这么做。它包含的几个文件常用于面向整个项目进行参数配置。

- settings.py：这个文件中包括了项目的初始化设置，可以针对整个项目进行有关参数配置，比如配置数据库、添加应用等。
- urls.py：这是一个 URL 配置表文件（或称为“模块”，在 Python 中，模块就是程序文件。此 URL 配置文件/模块，英文表示为 URLconf，即 URL configureation），主要将 URL

映射到应用程序上。当用户请求某个 URL 时，Django 项目会根据这个文件中的映射关系指向某个目标对象，该对象可以是某个应用中的 urls.py 文件，也可以是某个具体的视图函数。

- wsgi.py：WSGI 是 Web Server Gateway Interface 的缩写。读者可能听说过 CGI，CGI 是 Common Gateway Interface 的缩写，与 WSGI 有所不同。WSGI 是 Python 所选择的服务器和应用标准，Django 也会使用。wsgi.py 文件定义了我们所创建的项目都是 WSGI 应用。关于 WSGI 的更多知识可以参阅 http://wsgi.readthedocs.io/en/latest/index.html。
- __pycache__：如果创建项目后，不执行 python manage.py runserver 命令，不在浏览器中查看网站是否运行，那么它是不存在的。只有网站运行后，它才会出现。它其实是一个编译后的文件夹。看一下里面的文件，都是以.pyc 结尾的文件。关于编译问题，笔者已经在《跟老齐学 Python：轻松入门》中讲述过了，请查看相关章节。

3．blog

blog 是在项目中所创建的应用之一，注意是之一，用创建应用的指令还可以创建很多其他的应用。每创建一个新的应用，Django 就会在项目根目录（./）中创建一个子目录，并且目录中会有一些默认的文件。

- admin.py：在这个文件中，可以自定义 Django 管理工具，比如设置在管理界面能够管理的项目，或者通过重新自定义与系统管理有关的类对象，向管理功能增加新的内容。
- apps.py：这个文件是 Django 1.10 之后增加的，通常包含对应用的配置，比如为管理功能提供一个合适的应用名称。
- migrations：这是一个目录，用于存储应用的数据库表结构的指令，通过这些指令可以修改和创建数据库，从而在 models.py 模型类和数据库表之间迁移。
- models.py：这是应用的数据模型，每个 Django 应用都应当有一个 models.py 文件，虽然该文件可以为空，但不宜删除。
- tests.py：在这个文件中可以编写测试文档来测试所建立的应用。
- views.py：这是一个重要的文件，用于保存响应各种请求的函数或者类。如果编写的是函数，则称为基于函数的视图；如果编写的是类，则称为基于类的视图。views.py 就是保存函数或者类的视图文件。当然，也可以用其他的文件名称，只不过在引入相应函数或者类时，要注意名称的正确性，views.py 是我们习惯使用的文件名称。

4．db.sqlite3

这是一个默认的数据库。SQLite 是 Python 默认安装的数据库，在 Django 中也可以默认使用。因为到现在为止我们还没有配置数据库，所以就有了一个默认的文件（这个默认文件是在./mysite/settings.py 中配置生成的）。

至此，读者已经初步了解了 Django 项目的基本结构。若要让网站呈现纷繁多样的功能，还需要进行一些配置。

1.1.5 网站配置

前面仅仅是创建了应用 blog，还没有将该应用注册到本项目中，因此暂时不能访问。

在 Django 项目中，主管信息注册（对本项目进行各种信息声明）的文件是./mysite/settings.py。读者浏览这个文件时会发现里面的内容还不少，下面就选择几个目前会用到的进行说明。

- DEBUG：其值为 True 或者 False。在开发过程中，需要设置成 True，在调试（debugging，详见 https://en.wikipedia.org/wiki/Debugging）时，Django 能够显示详细的报错信息——这是“开发模式”。如果将项目部署到真正要对外发布的服务器上，我们称为“生产环境”，必须将其值修改为 False，从而避免暴露项目的内部信息。
- ALLOWED_HOSTS：在 DEBUG 为 True 时，其值可以为空。当部署到生产环境中时，要把主域名填写到这里，才能通过域名访问到本网站。
- INSTALLED_APPS：这是一个非常重要的配置项，所有的应用只有写到这里才能生效。在默认情况下，已经有了一些应用，比如 django.contrib.admin 就是针对项目后台管理的应用。现在需要把刚刚建立的应用 blog 配置到这里。在下面的 INSTALLED_APPS 列表中，①是新增加的，就是所建立的应用名称，其他各项是 Django 默认具有的应用。

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',    #①
]
```

- DATABASES：在这里可以配置数据库。Django 能够支持多种数据库，比如常见的 MySQL、PostgreSQL、Oracle 等。默认配置的是 SQLite，因为这个数据库小巧灵活，还是 Python 标准库所支持的，所以，本书中就使用这个数据库，便于读者使用和迁移代码，在实际的工程项目中，在服务器上可能很少用到它。如果用到了别的数据库，读者可以到官方网站查阅相关配置方式（https://docs.djangoproject.com/en/2.1/ref/databases/）。
- LANGUAGE_CODE：设置项目的语言，在一般情况下可以不用修改，如果非用汉语，则设置为 LANGUAGE_CODE = 'zh-hans'（注意不是'zh-cn'）。
- TIME_ZONE：设置时区，通常使用东八区，设置为“Asia/Shanghai”。

暂时简略说明上述几项，随着项目的发展，还会用到别的项目配置或者增加新的项目，届时再进行介绍。

至此，我们已经创建了项目和一个应用，并且对./mysite/settings.py 进行必要的配置。已经迈出了第一步，后面会更精彩。

1.1.6　知识点

1．开发模式

所谓“开发模式”，是相对“生产模式”而言的，即系统尚处于开发阶段，还没有正式对外部客户提供服务，在这种模式下很多配置都是为了开发而定的，比如在 Django 开发模式中，不需要配置 Apache 或者 Nginx 等服务器，也能够运行网站，这是因为 Django 本身就提供了简单的 Web 服务器功能，但是这仅限于开发过程，当网站被正式部署之后，即转换为“生产模式”时，就需要对部分配置进行修改。

在开发模式中，Django 会自动检测到修改的代码并重新加载，不需要每次修改代码后重新启动 Web 服务器。只有在新增加文件后，才需要重启 Django 服务。

运行 Django 服务的指令是 python manage.py runserver。

2．项目和应用

Django 安装好之后，就有了 django-admin 这个默认命令，可以用 diango-admin startproject projectname 命令创建一个 Django 项目。项目是由若干个“应用”（App）组成的，实现具体功能。创建应用可以使用 python manag.py startapp appname 命令，也可以使用 django-amdin startapp appname 命令。

创建了项目和应用之后，会生成一些默认的文件，它们要么是一些默认的配置，如 settings.py，要么是空文件，仅仅是为了占据一个位置而已，如应用里面的 views.py、models.py 等。

每个应用都要在项目的 settings.py 文件的 INSTALLED_APPS 中进行声明，告诉 Django 这个应用是本项目的一部分。

Django 将很多默认的功能（比如用户管理功能）视为一个应用。此外，Django 是一个开放的系统，任何人都可以开发第三方应用，这些第三方应用通常用于解决某个常见的问题，开发者在项目中使用第三方应用的主要目的就是节省时间、快速开发。这些第三方应用如果要在项目中使用，也要在 settings.py 的 INSTALLED_APPS 中注册后才能生效。

3．文档导读

（1）*Writing your first Django app*，https://docs.djangoproject.com/en/2.1/intro/tutorial01/。

（2）使用 Django 的网站，https://djangosites.org/。

（3）*Django FAQ*，https://docs.djangoproject.com/en/2.1/faq/。

1.2　编写博客的数据模型类

设计数据库和表结构是做网站的基础。在 Django 中，我们不需要通过 SQL 语句直接跟数据库打交道，而是完全用 Python 的方式创建数据模型，之后交给 Django 完成创建数据库的操作。读者不会使用 SQL，并不影响本书项目的学习。编写博客的数据模型类的基本知识如图 1-7 所示。

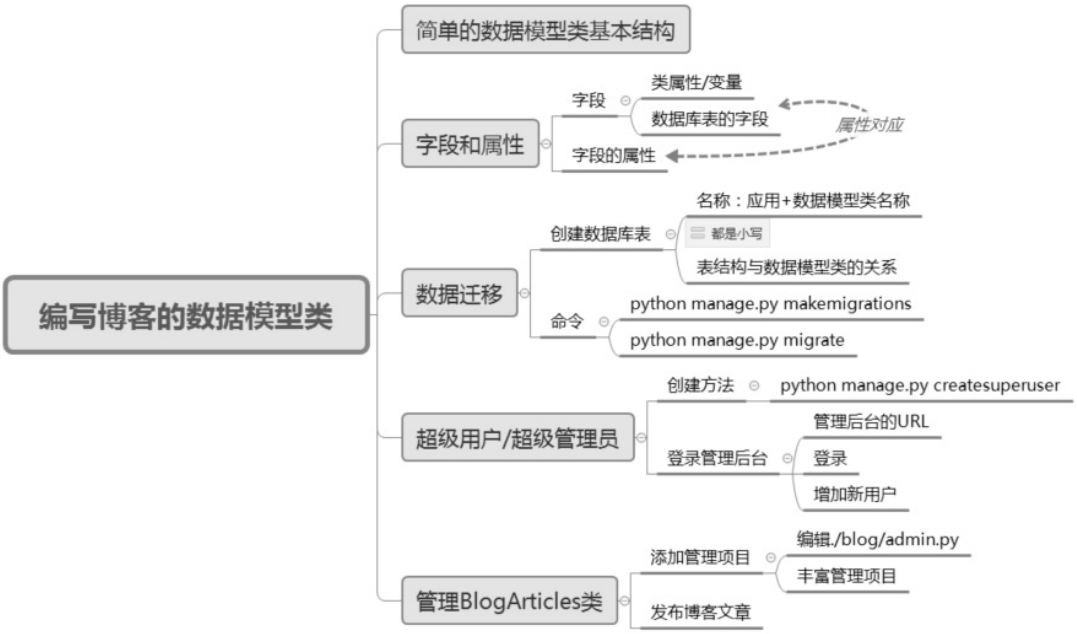

图 1-7　知识技能导图

1.2.1　数据模型类

利用 Django 开发网站系统，在一般情况下，要先编写数据模型，就是在./blog/models.py 中写一个类，这个类与数据库中的数据表具有对应关系。

别睡觉！这里需要你清醒。如果对 Python 中的类不熟悉，请及时复习，当然是参考《跟老齐学 Python：轻松入门》了。

下面就在./blog/models.py 中编写博客的数据模型类 BlogArticles。通过这个数据模型类，可以创建数据库中的一个专门用来保存博客文章的数据库表。

Django 中的数据模型类都继承自 django.db.models.Model 类。

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User

class BlogArticles(models.Model):
    title = models.CharField(max_length=300)    #①
    author = models.ForeignKey(User, on_delete=models.CASCADE,
                               related_name="blog_posts")    #②
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)

    class Meta:    #③
        ordering = ("-publish",)

    def __str__(self):
        return self.title
```

在上述这个类中，定义了一些属性，每个属性对应着将来数据库表中的一个字段。在后面的讲解中，提到数据模型类中的这些属性时，常常称之为字段。

语句①规定了字段 title 的属性为 CharField()类型，并且以参数 max_length=300 的形式说明字段的最大数量——这与用 SQL 语句创建数据库表时对每个字段做出类型规定很类似。

语句②通过字段 author 规定了博客文章和用户之间的关系——一个用户对应多篇文章，ForeignKey()就反映了这种“一对多”（或“多对一”）的关系。一个用户（用 OOP 的观点来看，一个用户就是 User 类的一个实例，在保存用户信息的数据库表中就是一条记录）对应于多篇文章（同理，一篇文章是 BlogArticles 类的一个实例，在相应的数据库表中是一条记录），通过这个字段，在“用户表”和“文章表”之间建立了关系。

官方网站中有对②中 ForeignKey()的完整参数说明（https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey），这里使用了其中的两个。on_delete 是必不可少的一个参数（对于 Django 2.x 版本），其值通常为 models.CASCADE，含义是“级联删除”。这是数据库操作中常用的一种操作。如果删除了“用户表”中的某个用户，因为“级联删除”的设置，“文章表”中该用户的文章记录也将被删除。当然，on_delete 的值还可以为其他，请参阅前述相关网页说明。

ForeignKey()中的另外一个参数 related_name="blog_posts"的作用是允许类 User 的实例（某个用户名）以“blog_posts”属性反向查询到类 BlogArticles 的实例。

语句③从名称上看貌似 Python 中的元类，但它跟元类不同。在此处，通过 ordering = ("-publish",)规定了 BlogArticles 实例对象的显示顺序，即按照 publish 字段值的倒序显示。

特别提醒读者，如果跟随本书进行练习，一定要注意观察，认真地输入每个字符。比如上面的代码中，ordering = ("-publish",)就很容易出错——元组中只有一个元素的写法，读者可以检验一下自己是否能够迈过这个“坑”。编程是一项细致的工作，需要仔细观察、耐心排查、反复尝试，方能成功。Web 开发又是各种知识的综合运用，如果读者对于 Python 基础知识有不熟悉之处，应该随时翻阅《跟老齐学 Python：轻松入门》。

BlogArticles 类的数据模型编写好了，将来数据库表的基本结构就是按照上述各字段及其属性而定的。如何根据数据模型建立数据库表呢？跟随下面的步骤继续操作。

在项目名称 mysite2 的根目录找到./manage.py，并执行 python3 manage.py makemigrations。

```
Chees-MBP:mysite2 qiwsir$ ls
blog     db.sqlite3    manage.py     mysite
Chees-MBP:mysite2 qiwsir$ python3 manage.py makemigrations
Migrations for 'blog':
  blog/migrations/0001_initial.py
    - Create model BlogArticles
```

注意观察上面的执行结果，提示信息中告诉我们在 blog/migrations 目录中创建了一个 BlogArticles 模型（Create model BlogArticles），不妨打开这个文件看一下（进入./blog/migrations 目录）。

```
Chees-MBP:migrations qiwsir$ more 0001_initial.py
# Generated by Django 2.1.4 on 2018-06-11 07:40
```

```
from django.conf import settings
from django.db import migrations, models
import django.db.models.deletion
import django.utils.timezone


class Migration(migrations.Migration):

    initial = True

    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
    ]

    operations = [
        migrations.CreateModel(
            name='BlogArticles',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name='ID')),
                ('title', models.CharField(max_length=300)),
                ('body', models.TextField()),
                ('publish', models.DateTimeField(default=django.utils.timezone.now)),
                ('author',
models.ForeignKey(on_delete=django.db.models.deletion.CASCADE,
related_name='blog_posts', to=settings.AUTH_USER_MODEL)),
            ],
            options={
                'ordering': ('-publish',),
            },
        ),
    ]
```

这个文件不是我们编写的，是在执行 python3 manage.py makemigrations 之后 Django 自动生成的。如果对上述代码的含义不是很清楚，还可以用下面的方法来看一下该文件的本质（退回到项目的根目录，即 manage.py 所在的位置）。

```
Chees-MBP:mysite2 qiwsir$ python3 manage.py sqlmigrate blog 0001
BEGIN;
--
-- Create model BlogArticles
--
CREATE TABLE "blog_blogarticles" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "title"
varchar(300) NOT NULL, "body" text NOT NULL, "publish" datetime NOT NULL, "author_id"
integer NOT NULL REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY DEFERRED);
CREATE    INDEX    "blog_blogarticles_author_id_ed798e23"    ON    "blog_blogarticles"
("author_id");
COMMIT;
```

这只是换一种方式来查看，如果读者熟悉 SQL 语句（如果不熟悉，可以先略过，等以后有机会再好好学习），就知道上述文件的功能是创建一个名称为 blog_blogarticles 的数据库表。这

个表的名称由两部分组成，第一部分 blog 是本应用名称的小写，第二部分 blogarticles 是在 models.py 中创建的数据模型类的名称的小写，中间用单下画线连接。这是 Django 中通过数据模型类创建数据库表通用的命名方式，即“小写的应用名称”+“_”+“小写的类名称”。

再观察数据库表中的字段名称，除 id 是自动生成外，其他都是在数据模型类 BlogArticles 中所声明的字段及其属性。

上面创建了一个能够建立数据库表的文件，下面就在此基础上，真正创建数据库了（注意操作指令时所在的目录位置）。

```
Chees-MBP:mysite2 qiwsir$ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying blog.0001_initial... OK
  Applying sessions.0001_initial... OK
```

本书中的项目使用 SQLite 数据库，并且在 setting.py 中规定了数据库文件存放在项目根目录中。

```
Chees-MBP:mysite2 qiwsir$ ls
blog  db.sqlite3  manage.py  mysite
```

对于 db.sqlite3 文件，可以使用专门的 GUI 工具软件打开，读者能搜索到很多。本书不使用 GUI 工具软件，而是在终端通过 SQL 语句查看数据库的有关内容，这需要读者具有 SQL 的基础知识，对于胸怀世界（hello world）的程序员而言，SQL 是不可或缺的。

```
$ sqlite3 /Users/qiwsir/Documents/Codes/DjangoPracticeProject/mysite2/db.sqlite3
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> .tables
auth_group                blog_blogarticles
auth_group_permissions     django_admin_log
auth_permission            django_content_type
auth_user                 django_migrations
auth_user_groups           django_session
auth_user_user_permissions
```

用 sqlite3 指令打开本项目所创建的数据库（如果读者的本地开发环境中没有配置 SQLite

环境，请自行到网上搜索配置方法）db.sqlite3 之后，即进入 SQLite 的操作终端，执行“.tables”即可呈现出当前数据库中已有的数据库表名称。除 blog_blogarticles 是刚刚通过 BlogArticles 模型建立的外，其他都是项目默认创建的数据库表。其他先不论，只看 blog_blogarticles，如下所示。

```
sqlite> pragma table_info(blog_blogarticles);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           title       varchar(30  1                       0
2           body        text        1                       0
3           publish     datetime    1                       0
4           author_id   integer     1                       0
```

上面的代码显示了数据库表 blog_blogarticles 的基本机构。仔细观察每个字段的数据类型，将这里的结果和前面在数据模型类 BlogArticles 中所规定的字段及其属性进行对比，进一步理解数据模型类中的各个属性含义。

这样就建立了博客的数据库——这是基础，接下来发布博客，并保存到这个数据库中。

1.2.2 发布博客文章

我们先使用 Django 默认的管理功能发布文章，这是最简单的方式。要使用此功能，必须先创建超级管理员。下面是创建超级管理员的方法，读者操作后要牢记自己所使用的用户名和密码。

```
$ python manage.py createsuperuser
Username (leave blank to use 'qiwsir'): admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

注意，在这里输入密码时，如果输入 123 之类的数字，系统会提示“This password is too short. It must contain at least 8 characters”；如果输入的密码太大众化，系统还会提示“This password is too common”。密码的设置要有一定的复杂程度，比如“1qazxcvbnm”。可见，Django 已经把安全问题想得很细致了，这也是笔者使用 Django 的重要理由。

牢记上面的超级管理员用户名和密码，然后运行 Django 服务。

```
$ python3 manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
June 11, 2018 - 17:03:50
Django version 2.1.4, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

在浏览器的地址栏中输入 http://127.0.0.1:8000/admin/，就可以打开如图 1-8 所示的界面。

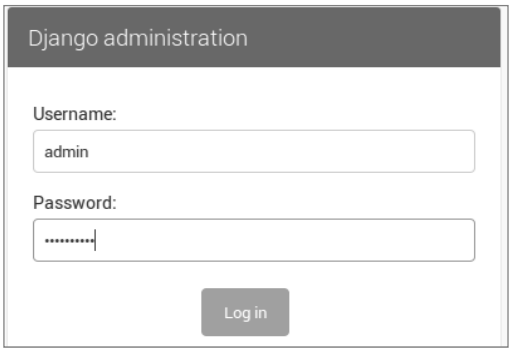

图 1-8　超级管理员的登录界面

输入刚才创建的超级管理员的用户名和密码进入系统，如图 1-9 所示。

图 1-9　超级管理员登录后的默认界面

Groups 和 Users 是 Django 在用户管理应用中默认的用户分类。单击 Users 会看到当前项目仅有的一个用户 admin，当然可以增加用户，读者一定要试一试。

目前暂不研究用户问题，前面已经说明了，目标是发布博客文章，结果登录进来之后没有找到发布文章的地方！

稍安勿躁，还记得前面介绍过的./blog/admin.py 文件吗？现在就要用到它了，用编辑工具打开它，然后输入如下代码。

```
from django.contrib import admin
from .models import BlogArticles    #①

admin.site.register(BlogArticles)    #②
```

代码①将 BlogArticles 类引入到当前环境，然后通过代码②将该类注册到 admin 中。

在调试状态下，如果没有新增加的文件，只是将原有文件修改了，则不需要重新启动 Django 服务（如果 Django 服务没有启动，请确保启动），然后将图 1-9 所示的页面刷新，得到如图 1-10 所示的结果。

图 1-10　增加了 BLOG 管理项

这时可以看到新注册的 BLOG。

单击 Blog articles 右侧的 Add 按钮可以添加博客文章，如图 1-11 所示。

图 1-11　admin 发布博客文章

在这个页面中输入一篇博客文章。请读者对照./blog/models.py 中的 BlogArticles 类，这里所填的表单中的各项是不是与 BlogArticles 类中的属性对应呢？

内容填好之后，单击页面右下角的 SAVE 按钮，该博客文章将被保存到数据库中，界面跳转到博客文章列表，并且在顶部提示此文章添加成功，如图 1-12 所示。

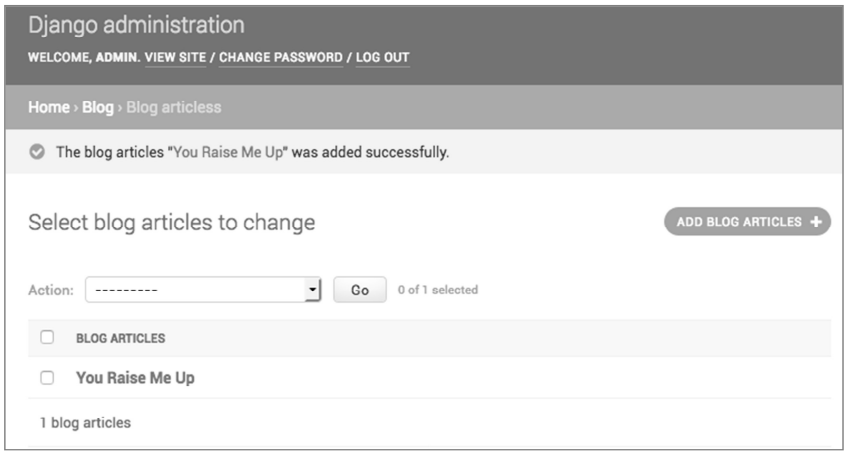
Django administration
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Blog › Blog articless
The blog articles "You Raise Me Up" was added successfully.
Select blog articles to change
ADD BLOG ARTICLES
Action:
Go
0 of 1 selected
BLOG ARTICLES
You Raise Me Up
1 blog articles

图 1-12　博客文章列表

如果你非常关心数据中是否有刚才所写的内容，可以用如下方式查看。

```
sqlite> .header on
sqlite> .mode column
sqlite> select * from blog_blogarticles;
id         title           body            publish             author_id
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
1          You Raise Me Up  When I am down 当我失意低落之时
and, oh my soul, so weary; 我的精神，是那么疲倦不堪
When troubles come 当烦恼困难袭来之际
and my heart burdened be; 我的内心，是那么负担沉重
Then, I am still 然而，我默默地伫立
and wait here in the silence, 静静地等待
Until you come 直到你的来临
and sit awhile with me. 片刻和我在一起  2018-06-11 09:11:33  1
```

从数据库中能够直观地看到刚才所保存的文档。注意，在发布文章时选择 admin 作为 author，并且 admin 在 Users 中的 id 是 1，所以这里记录了它的 id 值，即 author_id 为 1——这就是 BlogArticles 类中 ForeignKey()属性的效果。

为了后续操作需要，建议多发布几篇文章。

图 1-12 所示的列表信息太单一了。为了让列表页的信息丰富，还可以继续编辑./blog/admin.py，增加部分代码之后的文件内容如下。

```
from django.contrib import admin
from .models import BlogArticles

class BlogArticlesAdmin(admin.ModelAdmin):
    list_display = ("title", "author", "publish")
    list_filter = ("publish", "author")
    search_fields = ('title', "body")
    raw_id_fields = ("author",)
    date_hierarchy = "publish"
```

```
    ordering = ['-publish', 'author']

admin.site.register(BlogArticles, BlogArticlesAdmin)
```

修改./blog/admin.py 后进行保存，然后刷新浏览器页面，会呈现如图 1-13 所示的效果。

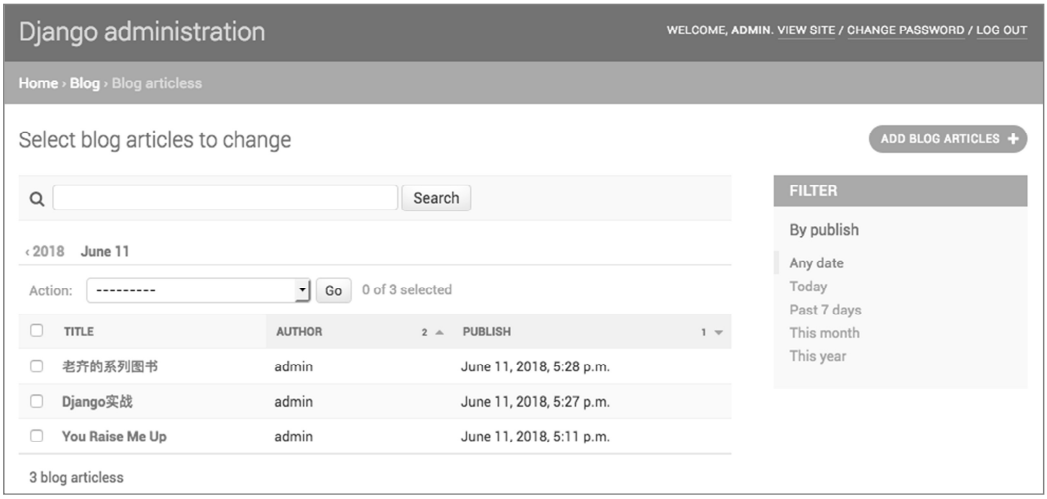

图 1-13　增加字段和管理项的列表页

至于文件中 BlogArticlesAdmin 各个属性的含义，读者可以用控制变量法依次研究，比如只保留第一个属性 list_display，将其他属性都注释掉，看看页面效果，就知道 list_display 的含义了——这是一种重要的方法，读者务必亲自尝试。

发布文章的目的是给别人看，别人怎么看？接下来解决这个问题。

1.2.3　知识点

1．什么是 HTTP

访问网站经常要用到 HTTP，HTTP 是 Hyper Text Transfer Protocol（超文本传输协议）的缩写，是客户端（比如计算机的浏览器、网络爬虫程序）和服务器端（网站）请求和应答的标准（TCP），封装了 Web 服务的整个过程。通常，由客户端发起请求，建立到服务器指定端口（默认是 80 端口）的 TCP 连接。服务器在指定端口监听客户端的请求，一旦收到，就向客户端返回状态信息（比如“HTTP/1.1 200”）和内容信息（如请求的文件、错误消息或者其他信息），这就是响应。简单地说，HTTP 要做的事情如下。

- 请求（request）：客户端到服务器端。
- 响应（response）：服务器端到客户端。

HTTP/1.1 协议共定义了 8 种请求方式，分别是 OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE 和 CONNECT。注意这些请求方式的名称都是大写，不要用小写。在本书的项目示例中，我们主要使用 GET 和 POST 两种请求，这也是 Web 网站经常使用的两种请求方式。

- GET：向指定的服务器发出请求，主要用于读取信息并显示。

- POST：向指定服务器提交数据，请求服务器进行处理（例如提交表单或者上传文件）。

随着技术的发展，现在另外一种协议正在被广泛使用——HTTPS，全称是 Hyper Text Transfer Protocol Secure，比 HTTP 多了一个 Secure。前面提到了 HTTP 默认的端口是 80，而 HTTPS 默认的端口为 443。相比 HTTP，HTTPS 安全性更高，因为 HTTP 以明文方式封装信息，如果被某些所谓的抓包工具获得，则很容易分析出其内容，而 HTTPS 以加密的方式传送信息，这也是它逐渐流行的原因。这里不对 HTTPS 做更多说明，有兴趣的读者可以自行查阅相关资料学习。本书示例的项目，暂且使用 HTTP。

2．什么是 URL

要在客户端访问服务器端，需要知道访问哪个服务器（服务器的位置），这就是 URL 要做的事情。URL 的全称是 Uniform / Universal Resource Locator，翻译为中文是“统一资源定位符”，俗称网页地址（网址）。它最初是由蒂姆 • 伯纳斯-李（Timothy John Berners-Lee）发明的，因为英女皇伊丽莎白二世在 2004 年向他颁发了大英帝国爵级司令勋章，所以常常称之为 Sir Timothy John Berners-Lee（蒂姆 • 伯纳斯-李爵士）。蒂姆 • 伯纳斯-李的发明为今天的互联网奠定了基础，而他也于 2017 年获得图灵奖。

有了 URL，就标志着步入了网络时代。

URL 的标准格式是：协议类型://服务器地址（必要时需加上端口号）/路径/文件名。

- 协议类型：可以是 HTTP，也可以是 HTTPS。
- 服务器地址：通常为域名，比如 itdiffer.com，也可以是 IP 地址。。
- 路径：以“/”区别目录。对于 GET 请求方式，还可以用“?”发起参数，每个参数以“&”隔开，再以“=”分开参数名称与值。
- 文件名：如果有必要可以写上，有时可以省略，由服务器根据路径匹配相应程序文件。

本书中的项目因为是在本地调试，所以可以使用 http://localhost:8000/path 或者 http://127.0.0.1:8000/path 的方式，读者也可以通过修改本地的域名，将 localhost 或者 127.0.0.1 替换为自己喜欢的域名（Ubuntu 系统请修改/etc/hosts 文件，Windows 系统请修改 system32/drivers/etc/host 文件）。

3．模型：ORM

在早期的网站中，都是把需要在网页上呈现的信息直接写到 HTML 文件中，这种类型的内容是静态的，现在网站中的某些部分依然存在这种静态网页，比如很多网站中的“关于本站”的简单介绍就是静态的。

随着网站功能的增加，需要为访问者提供动态内容。动态内容就是当用户发起访问请求时，网站实时地从数据库提取内容并呈现在网页上。在这类动态网站中，大多数是通过数据库实现对数据的保存和读取的，所以数据库是网站最基本和底层的组成部分。

Python 本身有读取数据库的模块（请阅读《跟老齐学 Python：轻松入门》有关内容），所以可以通过 SQL 语句直接实现程序和数据库的交互。Django 用另外一种方式解决了这个问题，在这种方式中不需要开发者使用 SQL 语句，而是使用更 Python 化的方式实现对数据库的操作，这就是 ORM，即 Object-Relational Mapping（对象关系映射）。

ORM 的作用是在关系型数据库和业务实体对象之间进行映射，这样在操作业务对象时，就不需要再去和复杂的 SQL 语句打交道，只需简单地操作对象的属性和方法。

Django 的数据模型层大量使用 ORM，这是一个仁者见仁、智者见智的事情，或者说这种处理方式并非都好，也有缺点。不过，在通常的开发中，这种方式能够给我们带来很大便利（它的缺点并不因此消失）。

- 可移植性强。ORM 通常具有很好的可移植性，本书中使用 SQLite 数据库，如果想改为 MySQL 数据库，只需要在 settings.py 文件中做好新数据库的配置，然后进行迁移数据的操作即可完成数据库的移植，不需要对 ORM 进行任何修改。
- 安全性好。使用 ORM 后很少或者不需要执行 SQL，所以这时就不必担心诸如 SQL 注入等形式的攻击了，更何况 ORM 还提供了一个自动引用和转义输入变量的机制，开发者不用在安全性的问题上花费太多时间，可以将精力集中在程序的业务逻辑和开发上。
- 查询语法简单。面对较为复杂的查询，如果使用 SQL 语句，常常要写很多；而使用 ORM，因为它本质上就是 Python 对象，能够让本来复杂的 SQL 语句变得简洁，所以能够实现更多的技巧。

上述三点，足以让我们至少在相当一部分项目中使用 Django 封装的这个数据模型层完成对数据库的操作。就中小项目而言，这种方式足够支撑业务需要，并且能够快速开发。

Django 的 ORM 表现方式就是编写数据模型类，这些类可以写到任何文件中，通常写在每个应用的 models.py 文件中。每个数据模型类都是 django.db.models.Model 的子类。应用的名称（小写字母）和数据模型类的名称（小写字母）共同组成一个数据库表的名称（“appname”_“model name”，例如 blog_blogarticles）。

在数据模型类写好之后，通过执行 Django 的数据迁移操作（python manage.py makemigrations，python manage.py migrate）就能够创建相应的数据库表，用来保存网站项目的数据。以后如果要修改数据库表的结构，只需要修改数据模型类，迁移数据就能够实现数据库结构的调整。所以，即使现在还不懂或者不太懂 SQL，也挡不住用户学习 Django 的步伐。当然，不能总不懂，以后还是应该懂一点数据库知识和相关语句。

4．文档导读

（1）*Object-relational mapping*，https://en.wikipedia.org/wiki/Object-relational_mapping。

（2）*What is an ORM and where can I learn more about it*，https://stackoverflow.com/questions/1279613/what-is-an-orm-and-where-can-i-learn-more-about-it。

1.3 显示博客信息

将博客内容保存到数据库还不是发布博客的终极目的，之所以要在网上撰写博客（曾经有人说是写自己的网络日记，这种说法有点骗人，要真的写自己的日记，肯定不会公开），目的就是通过吸引他人关注达到自己的某种目的，比如笔者在 www.itdiffer.com 网站上发布文章、炫耀自己的开发技能，目的就是吸引更多的读者来关注网站，进而促进图书的销售。所以，写博客一定要显示出来。显示博客信息的基本知识如图 1-14 所示。

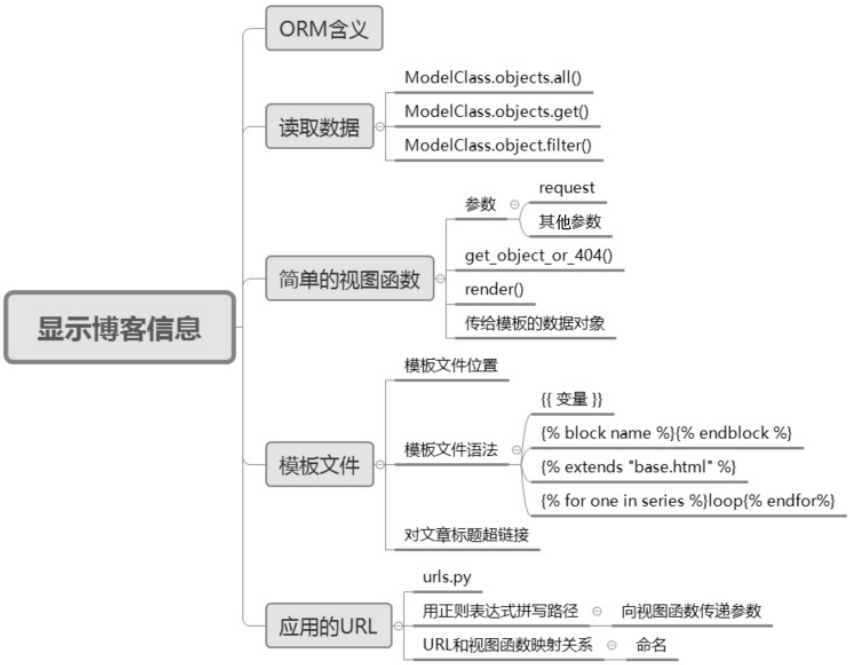

图 1-14　知识技能导图

1.3.1　显示文章标题

要显示文章标题，就要把标题从数据库中读出来。在数据库中专门有一个 title 字段存储文章标题，怎么才能读到它的具体内容呢？一种方法是通过 SQL 语句进行读取，另一种方法是使用 Django 进行读取。我们在./blog/models.py 中创建了数据模型后，Django 就会自动提供数据库抽象的 API，通过这个 API 可以创建、获取、修改和删除对象，对此还有一个专有的名称 ORM（Object-Relational Mapper）。

下面就在交互模式中练习一下对数据库进行增、删、改、查的操作。

```
$ python3 manage.py shell    #①
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 03:03:55)
Type 'copyright', 'credits' or 'license' for more information
(InteractiveConsole)
>>> from django.contrib.auth.models import User
>>> from blog.models import BlogArticles
```

请特别关注①，这里使用 python3 manage.py shell 的方式进入交互模式中，而不是用读者在学习基础知识时常用的 python 指令，因为用这样的方式已经将 Django 环境引入到了当前交互模式中。后面两行语句分别引入了两个对象，User 是 Django 默认的，BlogArticles 是我们在./blog/models.py 中写的。

```
>>> user = User.objects.get(username="admin")
```

上面这行语句的含义是获得 User 数据模型的字段 username 是 admin 的那个对象，或者说是数据库表 auth_user 中字段 username 的值是 admin 的那条记录（也是对象），因此变量 user 就是一个包含多个字段值的对象实例（如果用 Python 中有关类的知识来理解，可以认为是创建

了一个 User 类的实例，这个实例对象有一些属性和方法，下面的语句就是这个实例对象的属性）。

```
>>> user.username
'admin'
>>> user.id
1
>>> user.password
'pbkdf2_sha256$100000$pI606kU5dL8Z$ucytPawVJkC8sL4QViVvoli6BYtbGVZQBfsQ8CvlGZU='
>>> user.email
'admin@admin.com'
>>> type(user)
<class 'django.contrib.auth.models.User'>
```

对 BlogArticles 类进行类似操作，当然也可以将数据库中的所有记录读取出来。

```
>>> blogs = BlogArticles.objects.all()
>>> blogs
<QuerySet [<BlogArticles: 老齐的系列图书>, <BlogArticles: Django 实战>, <BlogArticles: You Raise Me Up>]>
>>> for blog in blogs:
...     print(blog.title)
...
老齐的系列图书
Django 实战
You Raise Me Up
```

貌似我们需要的东西已经被读取出来了——文章标题列表。是的，就是用这种方法，可以轻易读取到每篇文章的标题，其本质是一个由多个 BlogArticles 类的实例组成的序列对象。对于上述这种查询，在 Django 中被称为 QuerySet。

下面就利用刚才学习的知识写一个函数，专门用来读取文章标题，这个函数通常被写在./blog/views.py 文件中，当然也可以写到任何其他名称的文件中，但在本书中，我们遵此常规。

```
from django.shortcuts import render
from .models import BlogArticles

def blog_title(request):    #②
    blogs = BlogArticles.objects.all()    #③
    return render(request, "blog/titles.html", {"blogs":blogs})    #④
```

blog_title()是在视图文件 views.py 中编写的一个函数，这种方式我们称为“基于函数的视图”，这个函数被叫作“视图函数”。读者会在本书第 7 章看到另外一种“基于类的视图”的编写方式。这两种方式有着不同的应用场景，在学习了基于类的视图的编写方式之后，自然能够理解两者的差异。

刚刚编写的函数 blog_title()，是读者在本书中遇到的第一个响应用户请求的函数，其基本格式请特别关注。

blog_title 函数的参数是 request（如②中所示），这个参数负责响应所接收到的请求且不能缺少，并总是位于第一的位置。除这个不可或缺的参数外，还可以根据需要在其后增加别的参数。

语句③利用前面在交互模式中使用的语句，得到所有的 BlogArticles 类实例。

语句④以 return 结束当前函数，并返回结果。render()的作用是将数据渲染到指定模板上。还有另外一个函数 render_to_response()，也能够实现 render()的效果。但两者还是有所差别的：render()方法是 render_to_response()的快捷方式，会自动使用 RequestContext。render()的第一个参数必须是 request，然后是模板位置和所传送的数据，数据是用类字典的形式传送给模板的。

在 render()中出现的 blog/titles.html 就是展示标题列表的前端页面——被称为“模板”。在每一个应用中都可以有一个专门的模板目录，这个目录的名称必须是 templates。

现在进入应用 blog 的目录./blog 中，建立一个子目录 templates，名称和位置必须如此，再按照如下方式建立有关文件和子目录。

```
├─── templates
│    ├─── base.html
│    └─── blog
│         └─── titles.html
```

templates 目录是 Django 默认的存放本应用所需模板的目录，如果不用自定义的方式指定模板位置，Django 会在运行时自动来这里查找 render()函数中所指定的模板文件。

在模板目录中，有一个 base.html 文件，这个文件是将所有模板中公共的部分抽取出来，在其他文件中只需要编写个性部分的代码。也就是说，在 Django 的模板文件中，是可以有“继承”功能的。在本项目中，base.html 的代码如下。

```
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <meta charset="utf-8">
    <meta  name="viewport" content="width=device-width, initial-scale=1">
    <title>{% block title %}{% endblock %}</title>    #⑤
   <link rel="stylesheet" href="https://github.com/necolas/normalize.css/blob/master
/normalize.css">    #⑥
    <link rel="stylesheet" href="https://cdn.bootcss.com/bootstrap/3.3.7/css/
bootstrap.min.css">    #⑦
</head>
<body>
    <div class="container">
        {% block content %}    #⑧
        {% endblock %}
    </div>
<script src="https://cdn.bootcss.com/jquery/3.2.1/jquery.js"></script>    #⑨
<script src="https://cdn.bootcss.com/bootstrap/3.3.7/js/bootstrap.min.js">
</script>    #⑩
</body>
</html>
```

这是关于 HTML 的代码，如果读者尚缺乏相关基础知识，请自行补足，本书默认读者已经熟悉基本的 HTML/CSS/JavaScript 有关知识。

下面对这段代码的部分内容进行说明。

语句⑤和语句⑧都是在模板中定义了块。在 Django 模板中，{%block name %}是块标签，用它来包裹块内容，表示其间的内容可以自定义，name 是块的名称。语句⑤表示在模板中定义一个名为 title 的块。块的结束标签都是{% endblock %}。

语句⑥和语句⑦引入了样式表文件，只不过没有放在本地，而是通过网络地址引用远端的文件。语句⑨和语句⑩（<script>标签）则是引入了前端模板文件中常用的 JavaScript 库中的 jQuery 文件及有关的其他 JavaScript 文件。

这就是一个基础模板。下面编写与标题列表相对应的模板，即./templates/blog/titles.html 文件。

```
{% extends "base.html" %}    #⑪

{% block title %}blog titles{% endblock %}    #⑫

{% block content %}    #⑬
<div class="row text-center vertical-middle-sm">
    <h1>我的博客</h1>
</div>
<div class="row">
    <div class="col-xs-12 col-md-8">
    <ul>
    {% for blog in blogs %}    #⑭
        <li>{{ blog.title }}</li>    #⑮
    {% endfor %}
    </ul>
    </div>
    <div class="col-xs-6 col-md-4">
    <h2>广告</h2>
    <p>跟老齐学：www.itdiffer.com</p>
    <img width="200px" src=" https://raw.githubusercontent.com/qiwsir/
DjangoPracticeProject/master/python-book2.png">
    </div>
</div>
{% endblock %}
```

上面这段简短的代码包括了常用模板中的内容，下面分别说明。

语句⑪表示继承已经建立的模板 base.html，还可以理解为对 base.html 的扩展，即下面的代码都是基于 base.html 的，对其中的部分内容给予新的定义。

前面在 base.html 中已经定义了{% block title %}和{% block content %}块，语句⑫和语句⑬则是在本模板文件中对“父模板”base.html 中的同名称块标签进行重写。

在前面 blog_title()函数的语句④中，以{"blogs":blogs}的方式向本模板文件中传入了 blogs 变量所引用的 QuerySet 对象，其包含了所有 BlogArtilcles 类的实例对象，即从数据库表中读出来的所有记录（每条记录都可以看作是一个实例对象），并在模板文件中以变量 blogs（{"blogs":blogs}中的键 blogs）来代表。在本模板文件语句⑬中的{% block content %}{% endblock %}里面，将所有博客标题列表展现出来。

语句⑭是一个循环，有点类似 Python 中的循环语句，{% for blog in blogs %}是循环的开始，{% endfor %}表示本循环结束。

语句⑮中以{{ blog.title }}双层花括号的方式表示此处显示变量引用的数据。blog 是从 blogs 的 QuerySet 序列中得到的一个实例对象，title 是 BlogArticles 数据模型类的一个字段，blog.title 则表示某个实例的 title 字段，以这种方式得到一篇博客的（BlogArticles 类的实例）的 title 字段的值。

这里特别要提醒读者，注意④中指定用 title.html 显示博客名称列表的路径写法以及模板目录 templates 里面的结构。

函数和模板都编写好之后，要想通过网页访问，还需要做好另外一项重要的工作，就是进行 URL 配置。首先要配置./mysite/urls.py，在这个文件中配置本项目的各个应用。

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),
]
```

如果用户在浏览器的地址栏中输入了类似 http://localhost:8000/blog/的地址，通过这里的 URL 配置，将此请求转向 blog 应用的 urls.py，即./blog/urls.py。

在本书后续项目中，都是在./mysite/urls.py 中配置应用的 URL 后，再到某个应用的 urls.py 文件中配置该应用的具体 URL。

接下来就需要配置./blog/urls.py 中的 URL，当然这个文件还没有，需要先创建，然后配置如下 URL。

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.blog_title),
]
```

path()第一个参数为空，表示访问根，又因为在 blog 应用中，所以访问 blog 应用的根，即 http://127.0.0.1:8000/blog/。views.blog_title 声明了响应这个请求的函数（注意这里调用的是函数对象，不要写成 views.blog_title()的样式）。

重新启动 Django 服务。打开浏览器，输入地址 http://127.0.0.1:8000/blog/或者 http://localhost:8000/blog/，效果如图 1-15 所示。

图 1-15　博客标题列表页

显示文章标题的目的已经实现了，有了标题，就可以查看文章了。

1.3.2　查看文章内容

怎么才能查看文章内容呢？从功能上讲就是单击文章标题后呈现其详细内容。因此，文章标题还需要做超链接，对象就是文章详情页，对此应该在 titles.html 中显示标题的部分做相应修改。

```
{% for blog in blogs %}
    <li><a href="{{ blog.id }}">{{ blog.title }}</a></li>
{% endfor %}
```

再查看页面，发现每个标题都有了超链接，当然，如果此时单击，还不能显示其详情。要显示详细文章内容，还要做下面的工作。

在刚才的标题超链接中，是通过文章的 id 获得其详细内容的，对此可以先在交互模式下尝试一番。

```
>>> from blog.models import BlogArticles
>>> article = BlogArticles.objects.get(id=1)
>>> article.body
'When I am down 当我失意低落之时 \r\nand, oh my soul, so weary; 我的精神，是那么疲倦不堪 \r\nWhen troubles come 当烦恼困难袭来之际 \r\nand my heart burdened be; 我的内心，是那么负担沉重 \r\nThen, I am still 然而，我默默地伫立 \r\nand wait here in the silence, 静静地等待 \r\nUntil you come 直到你的来临 \r\nand sit awhile with me. 片刻地和我在一起'
>>> article.title
'You Raise Me Up'
>>> article.author
<User: admin>
>>> article.author.username
'admin'
>>> article.publish
datetime.datetime(2018, 6, 11, 9, 11, 33, tzinfo=<UTC>)
```

在交互模式中测试成功，接下来就可以放心大胆地将它们用在博客详情页的功能中了。

与前面的编写方法一样，还是要编辑./blog/views.py，增加响应查看文章详情请求的函数 blog_article()。

```
def blog_article(request, article_id):    #①
    article = BlogArticles.objects.get(id=article_id)    #②
    pub = article.publish
    return render(request, "blog/content.html", {"article":article, "publish":pub })
```

语句①中函数的参数列表比前面的 blog_title()多了一个参数，如何向函数传入这个参数呢？请阅读后续关于 URL 配置的相关内容。

语句②通过 article_id 参数创建一个 BlogArticles 类的实例，也可以说是读取 id 值是 article_id 的记录。

对比已经创建的两个函数发现其基本结构是一样的。使用基于函数的视图，都是按照这种方式编写函数的。

编写与之对应的模板，创建./templates/blog/content.html 文件，并输入如下代码。

```
{% extends "base.html" %}

{% block title %}blog article{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>{{ article.title }}</h1>
</div>
<div class="row">
    <div class="col-xs-12 col-md-8">
        <p class="text-center"><span>{{ article.author.username }}
            </span><span style="margin-left:20px">{{ publish }}</span>
        </p>
<div>{{ article.body }}</div>
</div>
    <div class="col-xs-6 col-md-4">
        <h2>广告</h2>
        <p>跟老齐学：www.itdiffer.com</p>
        <img width="200px" src="https://raw.githubusercontent.com/qiwsir/
DjangoPracticeProject/master/python-book2.png">
    </div>
</div>
{% endblock %}
```

接着配置 URL，因为还是针对 blog 这个应用而言，所以不需要修改./mysite/urls.py，只需要在./blog/urls.py 文件中增加新的 URL 路径。

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.blog_title),
```

```
    path('<int:article_id>/', views.blog_article),
]
```

在函数 blog_article(request, article_id)的参数列表中，除 request 外，还有 article_id，这个参数的目的是获得 URL 中每篇博客文章的 id，即./blog/templates/blog/titles.html 文件中的超链接<a href='{{title.id}}'>所产生的请求地址中最后的数字，例如 http://127.0.0.1/blog/1/。因此，使用<int:article_id>/这种方式，得到该请求的 id 数值并赋值给参数 article_id。

现在访问博客标题网页，然后查看某博客的详细内容（要养成习惯，每次测试效果时，都要查看 Django 是否启动了），如图 1-16 所示。

图 1-16　博客文章的详情页

注意观察网址，可以修改最后那个代表文章 id 的数字。笔者的示例项目中 id 最大值是 3，也就是说最多有 3 篇文章，如果修改为 5 或者其他值，这个页面就会变成如图 1-17 所示的模样了。

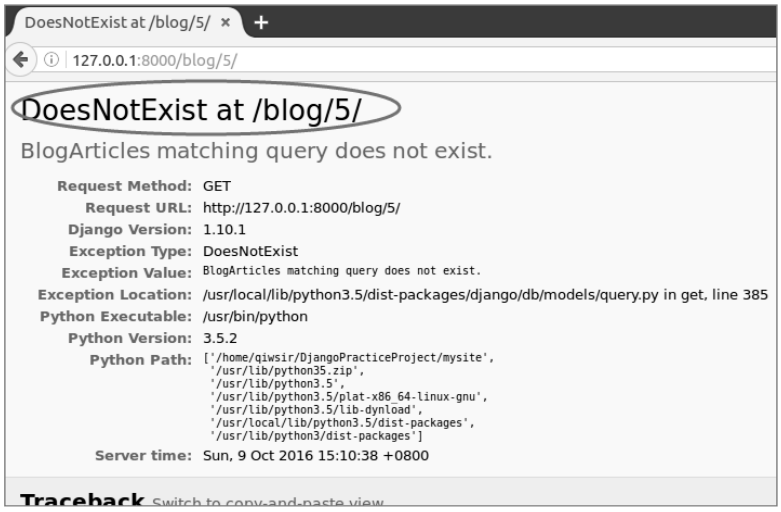

图 1-17　访问不存在的内容

因为不存在此 id，所以报错。我们现在处于 DEBUG 状态，显示了错误的完整信息。为了避免出现上述错误，应该在响应此请求的函数中对这种异常请求进行处理。所以，要适当修改./blog/views.py 中的 blog_article()函数。

```
from django.shortcuts import render, get_object_or_404    #①
from .models import BlogArticles

def blog_article(request, article_id):
    #article = BlogArticles.objects.get(id=article_id)
    article = get_object_or_404(BlogArticles, id=article_id)    #②
    pub = article.publish
    return render(request, "blog/content.html", {"article":article, "publish":pub })
```

语句① 引入了 get_object_or_404()方法，这个方法能够帮助我们简化对请求网页不存在时的异常处理。当请求对象不存在时，会抛出 DoesNotExist 异常。对于这个异常，可以用 try…except…捕获，在 except 中可以使用 raise Http404()来处理。这里我们使用 get_object_or_404()，省略了一些代码，此方法的参数列表为 get_object_or_404(klass, *args, **kwargs)。

- klass：一般是数据模型中的一个类（Model 类）。
- *args 和**kwargs：查询时的条件参数。

如语句②，使用 get_object_or_404()对 BlogArticles 依据 id 值进行查询，如果没有查询到任何数据，就返回 404 页面。

Django 中有很多默认方法，能够帮助我们提高开发效率，减少代码量。

再次请求图 1-17 所示的地址，就显示 404 错误了，如图 1-18 所示。

图 1-18　不存在页面的反馈

至此，一个博客就创建完成了，虽然简陋，但显示了 Django 在网站开发中的最基本结构。读者初次接触，要反复思考，可以将代码删除后凭借自己的理解单独做一遍甚至几遍，直到对此过程熟悉为止，从而为以后较复杂的开发奠定基础。“道路是曲折的，前途是光明的”，后续还有挑战。

1.3.3　知识点

1．Django 的 MTV

在动态网站开发中，普遍遵循 MVC 模式，这种模式最早由 Trygve Reenskaug 在 1978 年提出，是施乐帕罗奥多研究中心（Xerox PARC）在 20 世纪 80 年代为程序语言 Smalltalk 发明的一种软件架构。MVC 模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。

MVC（Model-View-Controller）模式就是把数据存取逻辑、业务逻辑和表现逻辑组合在一起的模式。在 MVC 模式中，Model 代表数据存取层；View 代表系统中选择显示什么和怎么显示的部分；Controller 是系统中根据用户输入和需要访问的模型，以决定使用哪个视图的部分。MVC 模式已经成为网站开发的共识，Django 更不例外，也遵循这种模式，或者说 Django 是一种 MVC 的框架，具体表现如下。

- M：数据存储和读取部分，由 Django 的数据模型部分承担。
- V：确定要显示哪些数据及如何显示，由视图和模板进行处理。
- C：根据客户端请求映射到相应的视图，由 Django 框架根据 URLconf 设置，调用相应的函数。

在 Django 中，Controller 部分事实上是由框架自动处理的，不需要开发者做什么，于是 Django 就转变为模型（Model）、模板（Template）和视图（Views）三部分了，可以称为 MTV 模式。

- M：模型（Model），即数据存取层，模型是网站项目的基础，主要负责处理与数据相关的事务，如读取、写入数据等。
- T：模板（Template），即表现层，处理与表现有关的事务，例如如何在页面中显示相关内容。
- V：视图（Views），即业务逻辑层，包含存取模型及调取相应模板的相关逻辑，是 M（模型）和 T（模板）之间的桥梁。在 Django 得到用户的请求后，根据 URL 映射关系调用相应的视图，视图则调用和处理有关数据。与模板相比，视图确定访问者能看到哪些数据，而模板确定怎么看到这些数据（或者说用什么方式看到这些数据）。

图 1-19 所示为 Django 对一个 HTTP 请求响应的流程图。

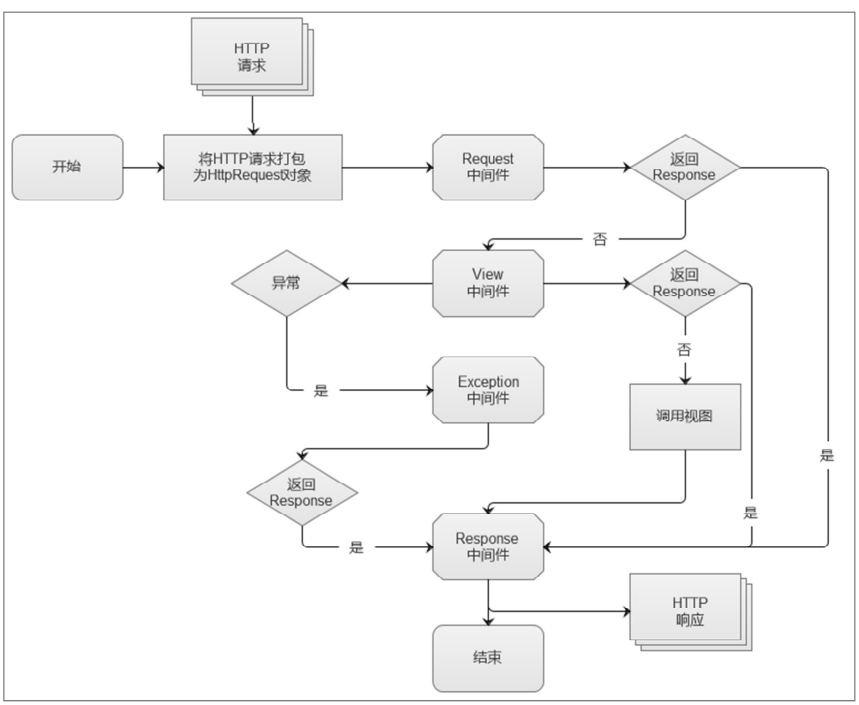

图 1-19　请求响应流程

从图中可以看到不少“中间件”，这是 Django 的核心，所有的请求、返回命令都由中间件来完成。这些中间件被定义在 settings.py 的 MIDDLEWARE_CLASSES 中。

2．Django 核心理念

Django 是一个开发框架，并且是有理念的开发框架，正是因为一直遵循这些理念，才使得它发展迅速，被众多开发者广泛接受。

- 更 Python 化。Django 是用 Python 编写的，它的很多约定也遵循 Python 的规范，有 Python 开发经验的用户能够很容易从事 Django 开发。
- DRY。这是很重要的原则，即 Don’t repeat yourself（不要重复自己）。这个原则可以作为所有程序开发的原则，但做起来真的有点难度。Django 中通过某些方式，在这方面做得相当不错，比如允许在数据模型中创建方法实现某些针对该模型实例的常见操作。读者在参照本书中的项目进行实践的过程中，要牢记该原则，并领会其含义。
- 松耦合与灵活操作。Django 使用 MTV 模式，实现了数据库访问、业务逻辑、模板显示等层面的松耦合和各自的灵活操作。在模板系统层面，如果用户不喜欢 Django 的模板系统，可以使用其他的模板系统；在数据库层面，用户可以自由地配置多种数据库，当然也可以不用 Django 封装的数据库 API，而是通过 SQL 语句直接操作数据库。不过，在本书的示例中，这些事情笔者都不做，只是告诉读者可以不遵循 Django 默认的任何东西，可以根据自己的需要自定义。
- 快速开发。这是非常重要的，Django 在不同层面上都提供了快速开发所需要的工具，比如通用视图，能够非常简单地组建一个网站。关于快速开发，在读者跟随本书项目逐渐展开的过程中，会有深刻体会。

上面列出了 Django 的几个核心理念，对于这些理念的理解，还需要读者在实践中体会。

3．URL 配置和查询

本节所讲解的 URL 配置，是在 Django 开发中常用的一种方式，读者在后续学习中要注意体会和应用。

当用户通过浏览器请求某个 URL 时，Django 会根据请求路径依次在 URLConf 中查询，并将第一个符合条件的映射关系作为查询结果。例如，访问 http://localhost:8000/blog/1/，其过程如下。

（1）localhost:8000：是主域名部分，不进行查询。

（2）/blog/：首先在./mysite/urls.py 中查询，遇到符合条件的 URL 映射（path('blog/', include('blog.urls')),），根据此映射中的描述，到 blog.urls（./blog/urls.py）中查询。

（3）/1/：在./blog/urls.py 中有 URL（path ('<int:article_id>/', views.blog_article) ）配置，请求的路径正好符合这里的正则表达式，从而确定了最终访问的视图函数 views.blog_ article。

这种 URL 的配置方法显然在多应用的项目中是比较适合的。如果是单应用的项目，可以直接在./mysite/urls.py 中配置（3）中所示的内容。

```
from django.urls import path
from . import views
```

```
urlpatterns = [
    path('', views.blog_title),
    path('<int:article_id>/', views.blog_article),
]
```

“<int:article_id>/”匹配的是一个数字字符，并且该数字字符赋给 article_id 参数，传给 views.blog_article()，这样 views.blog_article()函数除第一个参数 request 外，还有一个参数接收 article_id 传入的值。

对于 URLconf 中的视图函数的写法，笔者使用 views.blog_article 的方式，也可以写成字符串，如“views.blog_article”，两种写法没有优劣之分，但在同一个项目中要统一。

4．文档导读

（1）*URL*，https://en.wikipedia.org/wiki/URL。

（2）*URL dispatcher*，https://docs.djangoproject.com/en/2.1/topics/http/urls/。

第2章

用户管理

你是否删除了第 1 章创建的项目和博客应用？是否根据自己的记忆和理解又重建了第 1 章的项目和博客应用？如果没有，请不要开始本章。只有循序渐进、一步一个脚印、踏踏实实，方能成就旷世“武功”。

从本章开始，我们不再做单一用户的博客网站了，而是要一步一步地做一个多用户的文章管理系统。多用户，首要的是对众多用户进行管理，许可用户注册、登录、退出，并且超级管理员也能够对所有用户进行管理。“不积跬步，无以至千里；不积小流，无以成江海。”任何一个大系统都是由许多小功能组成的。

2.1 自定义模板和静态文件位置

依然使用第 1 章中创建的项目，不过，因为是大一点的系统了，所以会创建多个应用，在这种情况下，模板、静态文件等都要指定放在某个位置。笔者不赞成将模板和静态文件分散到各个应用内，虽然这样也可以，但在这里，即使读者对笔者的观点心存异议，也请保留，因为笔者还有另外一个目的，就是介绍如何在./mysite/settings.py 中设置模板和静态文件的位置，以备后用。自定义模板和静态文件位置的基本知识如图 2-1 所示。

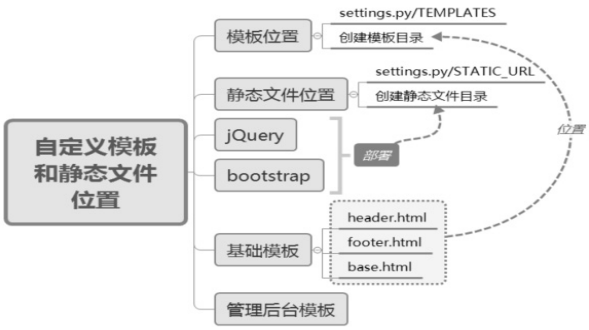

图 2-1　知识技能导图

2.1.1 自定义模板位置

编辑./mysite/settings.py 文件，修改 TEMPLATES 的值。

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates'),],    #①
        'APP_DIRS': False,    #②
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

语句①定义了模板文件的位置，并且该目录的名称 templates 也在这里定义了，BASE_DIR 表示本项目的根目录，用 os.path.join()函数将两者连接起来，即模板目录的位置是相对项目根目录的 templates 目录——./templates。显然，读者可以按照自己的意愿来定义模板目录的名称和位置。

语句②将 APP_DIRS 设置为 False，即不再允许 Django 按照默认方式寻找模板文件。

根据上述配置要求，在网站根目录建立 templates 目录。

```
Chees-MBP:mysite2 qiwsir$ mkdir templates
Chees-MBP:mysite2 qiwsir$ ls
blog     db.sqlite3   manage.py    mysite       templates
```

重新执行 python3 manage.py runserver，在浏览器的地址栏中输入 http://localhost:8000/blog，是什么结果？是不是报错了？

肯定是的。

因为我们仅建立了一个空文件夹，里面根本没有模板。将原来放在./blog/templates/中的文件和目录移动到./templates 中，之后再刷新页面，就看到熟悉的“脸庞”了。

从现在开始，后续应用所用的模板都放在./templates 中。

2.1.2 自定义静态文件位置

在网站开发中，常常将网页中的 CSS、JavaScript 文件及网页上的图片称为静态文件——与被渲染的模板文件比较。

打开./mysite/settings.py 文件，在最后几行中，会看到如下代码。

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/2.1/howto/static-files/

STATIC_URL = '/static/'
```

先看上面两行注释的含义，第一行说明了 Django 中认定的静态文件是什么，第二行告诉我们想要知道怎么做可以去查看一个地址。最后一行代码 STATIC_URL = '/static/'的意思是在默认状态下，如果通过 URL 访问静态文件，则可以通过（以刚刚建立的 blog 引用为例子）如 http://localhost:8000/blog/static/newton.jpg 的模式进行访问，即在./blog 目录中建立一个名为 static 的目录，然后将静态文件（如 newton.jpg）放在里面。

这种方式是让每个应用都有自己的静态文件存放地。但是，本项目中不采用这种方式，而是用类似 2.1.1 节中对模板文件管理的方式，把所有静态文件都放到指定的目录中。因此，继续编辑./mysite/settings.py 文件，在 STATIC_URL = '/static/'的下面增加如下代码。

```
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static"),
)
```

通过上述配置，指定了静态文件存放目录的位置，即在本项目根目录中创建./static 子目录。为了进行后续测试，在./static 目录中放置一个图片文件（如 newton.jpg 文件）。

```
$ ls
blog  db.sqlite3  manage.py  media  mysite  static  templates
```

完成上述操作之后，需要重新启动 Django 服务，用浏览器再次打开 http://localhost:8000/static/newton.jpg，是否访问到静态文件了？

之所以能够如此设置，是因为 Django 的 settings 配置变量中的 STATICFILES_FINDERS 的默认值规定了静态文件的查找顺序和内容。

```
STATICFILES_FINDERS = (
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder"
)
```

按照这个默认值，首先读取 STATICFILES_DIRS 中规定的静态文件（django.contrib.staticfiles.finders.FileSystemFinder），如果没有设置此值，就找不到什么，但不会对网站运行造成影响。然后通过 django.contrib.staticfiles.finders.AppDirectoriesFinder 在每个应用中查找有没有默认的静态文件目录 static。如果读者在命名时将两处都做了设置，而且静态文件的名字还都一样，那么 Django 会在找到第一个后终止查询。

建议读者在上面例子的基础上验证此说法。

在实际项目中，通常要将所有的静态文件都归到同一个目录，为了区分不同应用的静态文件，通常还会在里面再建立子目录，后续会看到此类操作。

2.1.3　通用静态文件和基础模板

在第 1 章的示例中，已经使用了 bootstrap.css 静态文件，但是这个静态文件是放在别人的服务器上的，我们通过网址引用。这样做虽然简单，却不能保证对方永久提供该链接。在一般情况下，我们选择大的服务商提供的文件，就假装相信它们能永久提供了。

在某种心理作用下，还是觉得将文件放在自己的服务器上放心，于是就有了下述内容。

所谓通用静态文件，就是会在很多应用中使用到的文件，比如 jquery 和 bootstrap 相关的文件。

先到各自官方网站上下载相应的文件，然后放到已经建立的 static 目录中。为了区分不同类型的文件，要分门别类地建立目录，目录结构如下。

```
./static$ tree
├─── css
│    ├─── bootstrap.css
│    ├─── bootstrap.css.map
│    ├─── bootstrap.min.css
│    ├─── bootstrap-theme.css
│    ├─── bootstrap-theme.css.map
│    └─── bootstrap-theme.min.css
├─── fonts
│    ├─── glyphicons-halflings-regular.eot
│    ├─── glyphicons-halflings-regular.svg
│    ├─── glyphicons-halflings-regular.ttf
│    └─── glyphicons-halflings-regular.woff
├─── images
└─── js
    ├─── bootstrap.js
    ├─── bootstrap.min.js
    ├─── jquery.js
    └─── npm.js
4 directories, 14 files
```

这样就将通用的部分静态文件存放好了。还有一些图片，比如网站的 LOGO，可以放到 images 里面。

这些通用的静态文件常常在基础模板中使用，比如第 1 章示例中的 base.html。针对本章所要创建的项目，我们要重新制作基础模板。

一般来讲，网站的界面大致可以分为上、中、下三个部分。上部就是头部，显示网站的 LOGO 和导航等；中部就是要显示的具体内容；下部就是底部，显示网站的版权信息或者诸如联系方式、购买流程等不怎么变化的信息或超链接。

按照上面的划分，头部和底部都可以做成基础模板。

在./templates 目录中创建 header.html 文件和 footer.html 文件，header.html 文件代码如下。

```
{% load staticfiles %}    #①
<div class="container">
    <nav class="navbar navbar-default" role="navigation">
        <div class="navbar-header">
            <a class="navbar-brand" href="http://www.itdiffer.com">
               <img src="{% static '/images/logo.png' %}" width="100px">
            </a>    #②
        </div>
        <div>
            <ul class="nav navbar-nav" role="navigation">
```

```
                <li><a href="{% url 'blog:blog_title' %}">BLOG</a></li>    #③
            </ul>
            <ul class="nav navbar-nav navbar-right">
                <li><a href="#">LOGIN</a></li>
            </ul>
        </div>
    </nav>
</div>
```

这就是所谓的头部代码。

语句①是模板中声明引入静态文件的标签，只有使用它，语句②（<a>标签）中才能使用{% static 'images/logo.png' %}，而不是用“硬编码”的方式写图片地址。语句②中的这个写法也是 Django 模板的一个语法，{% static %}可以用来引入某个静态文件，此处表示引入了静态文件中的一张图片，注意使用格式和文件路径。语句①写在语句②前面即可，此处之所以写在顶部，纯属编写习惯，没有特别规定。

语句③中的{% url 'blog:blog_title' %}试图实现的是对 blog 应用的访问，实现对“BLOG”字样的超链接“http://localhost:8000/blog/”。

- url：Django 模板内置的标记符，表示在此处返回由应用和参数组成的 URL。
- 'blog:blog_title'：URL 的名称，其中 blog 是应用名称，blog_title 是应用中的某个 URL 的名称。

为了实现{% url 'blog:blog_title' %}的效果，需要在项目和应用两级的 urls.py 文件中进行相应的配置。Django 中有两种配置方式（以下两种方式适用于 Django 2.x）：

方式一：

- 在./mysite/urls.py 的 urlpatterns 中进行如下设置。特别注意，include()中的第一参数提供的是一个二元的元组。

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include(('blog.urls', 'blog'), namespace='blog')),
]
```

- 在./blog/urls.py 的 urlpatterns 中进行如下配置。

```
urlpatterns = [
    path('', views.blog_title, name='blog_title'),
    path('<int:article_id>/', views.blog_article, name='blog_article'),
]
```

方式二：

- 在./mysite/urls.py 的 urlpatterns 中进行如下设置。注意，被注释了的那一行是“方式一”的配置，请读者注意比较两种配置方式之不同。

```
urlpatterns = [
    path('admin/', admin.site.urls),
    #path('blog/', include(('blog.urls', 'blog'), namespace='blog')),
    path('blog/', include('blog.urls', namespace='blog')),
]
```

- 在./blog/urls.py 的 urlpatterns 中进行如下配置，比“方式一”增加了 app_name = "blog" 这一行。

```
app_name = "blog"

urlpatterns = [
    path('', views.blog_title, name='blog_title'),
    path('<int:article_id>/', views.blog_article, name='blog_article'),
]
```

以上两种方式，供读者自行选择，任何一种都能实现语句③的目的，这样就避免在语句③中“硬编码”了。

再看 footer.html 的代码，就相当简约了。

```
<div class="container">
    <hr>
    <p class="text-center">copy right www.itdiffer.com</p>
</div>
```

头尾代码编写完之后，把它们放到一个名为 base.html 的文件中进行整合，这个文件在前面已经创建了，此处结合刚才编写的 header.html 和 footer.html 文件，将 base.html 文件重写。

```
{% load staticfiles %}
<!DOCTYPE html>
<html lang="zh-cn">
    <head>
     <meta charset="utf-8">
     <meta name="viewport" content="width=device-width, initial-scale=1">
     <title>{% block title %}{% endblock %}</title>
     <link rel="stylesheet" href="{% static 'css/bootstrap.css' %}">
    </head>
    <body>
        {% include "header.html" %}
        <div class="container">
          {% block content %}{% endblock %}
        </div>
        {% include "footer.html" %}
        {% block javascript %}{% endblock %}
    </body>
</html>
```

关于 blog 应用本身的模板就不需要做任何修改了。如果此前 Django 服务器在运行，那就就按 Ctrl+C 组合键退出，然后再次运行，打开浏览器，在地址栏中输入 http://localhost:8000/blog/，效果如图 2-2 所示。

是不是感觉很神奇？同时觉得自己很厉害？程序员最容易自我膨胀了，直到遇到 Bug 之后就陷入无限的沮丧，然后奋起捉“虫子”，哪怕是修改了愚蠢的错误，也会再次欣喜若狂。这就是程序员，在起起伏伏中生活着。

图 2-2　博客列表页

小心 Bug！

2.1.4　重置管理后台模板

还记得第 1 章中我们在 Django 默认的管理后台看到的有关用户管理的项目吗？为了加深印象，在浏览器的地址栏中输入 http://127.0.0.1:8000/admin/（确认 Django 服务已经运行），按 Enter 键之后，会被打开的界面所惊呆，大出意外，如图 2-3 所示。

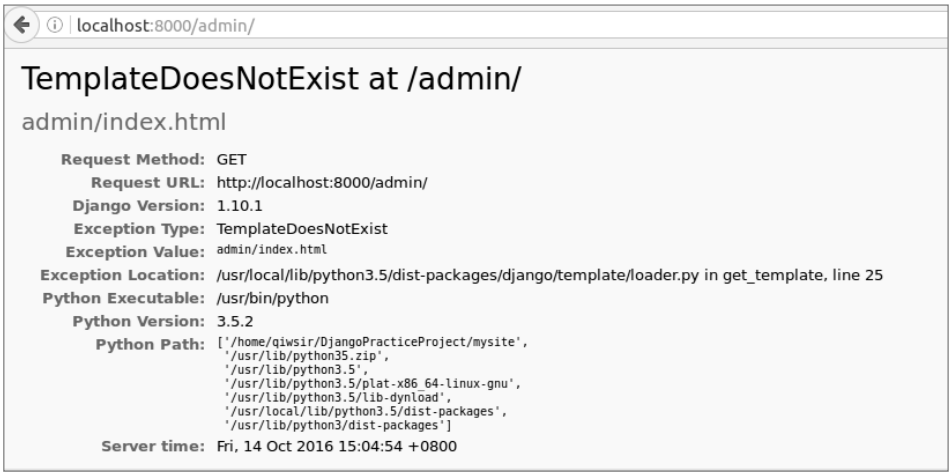

图 2-3　超级管理员模板不存在

“TemplateDoesNotExit”模板不存在，为什么会这样？关于后台管理部分，我们并没有进行修改呀！是不是 Django 出问题了？还是我的操作系统有问题了？或者谁修改了我的程序？我可以断定，肯定不是我的错（程序员面对 Bug 时的第一反应都是如此：肯定不是我的错！不可能有 Bug）！

少安毋躁。

当一个程序员面对各种错误时，很容易产出疑问，甚至找借口来说明异常的出现不是自己

的错，这不是优秀程序员的修养。当然，人们在很多事情上都会采用类似的思维方法。

“为将之道，当先治心。泰山崩于前而色不变，麋鹿兴于左而目不瞬，然后可以制利害，可以待敌。”（苏洵《心术》）这是一个有志于改变世界的程序员应有的修养。作为一个有修养的程序员，要认真思考导致异常的原因，找到解决之道。

查找原因的方法就是梳理从上次可以访问到现在不能访问的这段时间中，我们都做了哪些修改。回头审查一下操作过程，不难发现，我们已经将默认的文件模板位置都指向了自定义的模板位置，再认真看界面上的异常信息，在上述截图的后面还有一句话，不能丢掉，如下所示。

```
Template-loader postmortem
Django tried loading these templates, in this order:
Using engine django:
    django.template.loaders.filesystem.Loader:  /Users/qiwsir/Documents/Codes/
DjangoPracticeProject/mysite2/templates/admin/login.html (Source does not exist)
```

我们自定义的模板在网站根目录的./templates 中，但是，到目前为止我们还没有将后台管理所对应的模板放到这里，在原来的 Django 配置中，并没有显明后台管理对应的模板位置，这说明它应该在默认的某个秘密位置。

在 2.1.1 节中，为了让模板指向./templates，修改了./mysite/settings.py 文件中 TEMPLATES 的两个值——DIRS 和 APP_DIRS。其中，APP_DIRS 的值为 False，就使得 Django 在搜索模板的时候不再搜索应用中的 templates 目录，所以导致了刚才的错误。

原因找到了，解决方法就明朗了。将./mysite/settings.py 中 TEMPLATES 的 APP_DIRS 的值修改为 True。修改之后，Django 搜索模板的路径就变成：首先根据 DIRS 中规定的路径搜索，如果找不到，再到每个应用的 templates 目录中搜索。

完成上述修改之后，再次刷新 http://127.0.0.1:8000/admin 页面，完成登录之后，就能看到如图 2-4 所示的界面。

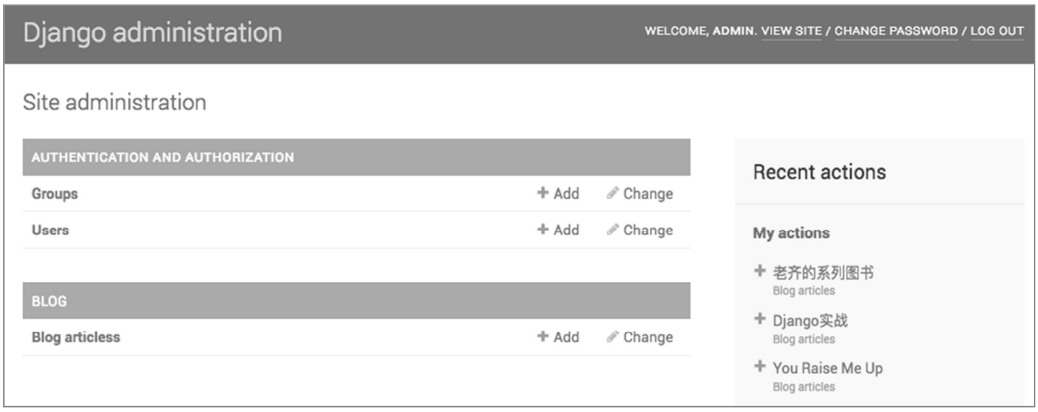

图 2-4　管理界面

终于可以长出一口气了。平复一下刚才激动的心情，此情此景，真想赋诗一首：

本想仗剑走天涯，

怎奈 Bug 多。

山重水复疑无路，

蓦然回首，

虫儿被捉住。

2.1.5　知识点

1．模板

前面已经在开发过程中用到模板了，这里对相关知识进行总结归纳。Django 中的模板本质上也是一个独立的文本文件（其实不仅是 Django，其他的 Web 框架模板也是如此），文件中包含了静态内容（比如 HTML、CSS）和动态标记的数据。Django 中的视图函数（后面会学习另外一种基于类的视图，在那种方式中就是类）决定了使用哪个模板及向模板渲染什么数据。

编写模板所用的语言以 HTML、CSS、JavaScript 为主，作为一名 Django 开发者，不要认为这是前端的事情而自己可以不懂，现在有价值的是“全栈”，虽然会前端不意味着“全栈”，但毕竟比仅会后端更靠近全栈的要求了。模板不是嵌入 Python 语言，但是为了增强其表现力，里面也会通过一些模板语言来实现类似循环那样的逻辑。

2．contexts

contexts 翻译成什么？上下文管理？这种翻译有点蹩脚，因为翻译过来之后更不理解原来的意思了。在没有找到更好的翻译词语之前，笔者暂时不翻译，直接使用这个单词，读者可以把它看作一个专有名词。

模板是一个动态文档，只有在显示动态信息时才能感受到它的存在。Django 的视图函数在渲染模板（将有关数据传给模板）时使用的中介物就是 contexts。contexts 是一个包含了键值对的字典，例如前面曾经使用过的 return render(request, "blog/content.html", {"article":article, "publish":pub })，这条代码通过 render()函数声明了所使用的模板（"blog/content.html"）和 contexts（{"article":article, "publish":pub }）。在 contexts 的键值对中，“键”是即将在模板中使用的变量，这个变量可以引用任何在视图函数中确定的对象。

3．模板：for 和 if

所谓模板语法，就是根据 Django 的规定，在模板中使用通过 contexts 传入模板的数据对象。一般情况下有两类语法，一类是单独的变量，也称为变量标签；另一类是块命令，也称为块标签。

例如，前面使用过的{{ publish }}就是变量标签，publish 是 contexts 中的一个“键”，通过它将 contexts 键值对中的值传入模板，即在该位置显示所渲染的数据内容。注意，这种变量标签必须用{{ variable }}的形式——使用双花括号（大括号）。

除变量标签外，还有“块标签”，下面的代码实现 for 循环，就是一种块标签。

```
{% for blog in blogs %}
   <li><a href="{{blog.id}}">{{ blog.title }}</a></li>
{% endfor %}
```

块标签以{% command %}开头（{% for blog in blogs %}），并以{% endname %}结尾（{% endfor %}），中间是块的内容，上例中的{{ blog.title }}依然是变量标签，只不过这个变量的内

容来自 contexts 的键 blog 引用的对象的一个属性（title）。

对于 for 循环标签，在模板中应用时，常常会使用循环计数功能，为此 Django 模板提供了默认的{{ forloop.counter }}从 1 开始计数的方式，不需要视图函数中的 contexts 提供 forloop 键值对，只要是模板中的循环，都可以使用这样的方式实现对被循环内容的计数功能。另外，还有{{ forloop.counter0 }}是从零开始计数的。

在模板中除可以实现 for 循环外，还可以实现 if 结构，其样式如下。

```
{% if condition %}
    something
{% endif %}
```

注意，在 Django 模板的 if 标签中也有类似 Python 的 if…else 结构。

```
{% if condition %}
    something
{% else %}
    others
{% endif %}
```

以上两种块标签在模板中比较常用，而且它们也能够被嵌套使用。

读者暂且对模板的语法有初步了解即可，随着项目的进行，还会涉及新的模板语法，甚至还可以自定义模板标签。

4．文档导读

（1）*Managing static files*，https://docs.djangoproject.com/en/dev/howto/static-files/。

（2）*The Django template language*，https://docs.djangoproject.com/en/2.1/ref/templates/language/。

（3）*Built-in template tags and filters*，https://docs.djangoproject.com/en/2.1/ref/templates/builtins/。

2.2 用户登录

前面我们创建了一个超级管理员用户 admin（当然也可以命名为其他名称），并且可以通过管理员登录页面（http://localhost:8000/admin/）登录到管理后台，而且在这个后台发布了博客。同时，读者也一定关注到，这个后台中有用户管理功能，如图 2-4 所示。

通过后台管理界面中的 Users 可以向本网站中增加用户，如图 2-5 所示就是增加几个用户之后的 Users 列表。

但是，除 admin 外，其他用户是不能轻易登录管理后台的。对于一个多用户的文章系统，非管理员用户最好是从前台登录，登录之后仅具有所规定的功能。

USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
admin	admin@admin.com			✓
wanglaoshi	wang@laoshi.com	wang	laoshi	✗
zhanglaoshi	zhang@laoshi.com	zhang	laoshi	✗

图 2-5　用户列表

下面首先解决登录问题，所需要的知识技能如图 2-6 所示。

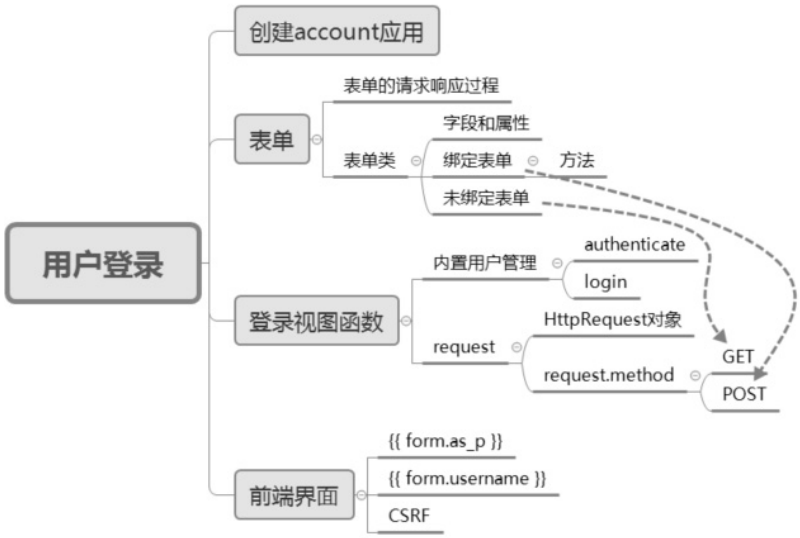

图 2-6　知识技能导图

2.2.1　创建应用

为了实现用户登录、退出、注册等功能，从而进行用户管理，我们要创建一个新的应用。

```
$ python3 manage.py startapp account
$ ls
account  blog  db.sqlite3  manage.py  mysite  static  templates
```

创建新应用的表现就是在项目根目录中多了一个 account 目录，这与前述创建 blog 应用是一样的。

之后依然是在./mysite/settings.py 中对新应用进行配置。

```
INSTALLED_APPS = [
    … ,
    'blog',
    'account',
]
```

接着就是在./mysite/urls.py 中进行 URL 配置。

```
urlpatterns = [
    … ,
    path('account/', include('account.urls', namespace='account')),
]
```

最后在./account 目录中创建 urls.py 文件，并设置好本应用中的路径。

```
from django.urls import path
from . import views

app_name = "account"
```

```
urlpatterns = [
    path('login/', views.user_login, name='user_login'),
]
```

views.user_login 意味着必须要在 views.py 中创建一个名为 user_login 的函数来响应请求。至此，基本配置已完成，下面开始设计用户登录过程。

2.2.2 理解表单类

在很多网站中，都有用户登录框（暂不讨论新兴的扫二维码登录），用户登录框用行话说就是表单。这个表单可以用纯粹前端的 HTML 代码来编写，也可以用我们下面要介绍的表单类来编写。这两种方式在实践中都可以使用，这里我们首先应用表单类。

在./account 目录中新建文件 forms.py，这个文件专门存放各种与表单有关的类（文件的命名也是依据惯例），代码如下。

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

为了理解这个表单类，下面在交互模式中进行操作，一步一步地试验。在网站项目的根目录执行“$ python3 manage.py shell”，进入交互模式中。

```
>>> from account.forms import LoginForm
>>> login_form = LoginForm()
>>> print(login_form)
<tr><th><label  for="id_username">Username:</label></th><td><input  id="id_username"
name="username" type="text" required /></td></tr>
<tr><th><label  for="id_password">Password:</label></th><td><input  id="id_password"
name="password" type="password" required /></td></tr>
```

用 from account.forms import LoginForm 引入./account/forms.py 中建立的表单类 LoginForm，然后创建这个类的实例，只不过没有向类中传递任何参数，这个实例可以称为未绑定（数据）实例，然后打印此未绑定实例，结果就是 HTML 代码。这些 HTML 代码构成了前端表单，其对话框与我们在./account/forms.py 中所写的 LoginForm 类的属性是对应的。因此，可以认为不论是前端的表单，还是后端的表单类，都是对象。这个对象的属性就是表单中的对话框（input）。

下面分析细节部分。

类 LoginForm 中的 username = forms.CharField()对应于交互模式中打印出来的<input name='username' type='text' …>，而在 password = forms.CharField(widget=forms.PasswordInput)中，因为使用了 widget=forms.PasswordInput，故其对应<input type="password" … >。在表单类的属性值中，通常使用 widget 来规定相对应的 HTML 元素的类型，如本例中规定 input 的类型为 password。

对象除了属性，还有一些方法。比如表单类的未绑定实例 login_form，可以使用 Python 中已经熟知的自省方法查看其方法。

```
>>> dir(login_form)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
```

```
'__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__html__',
'__init__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_bound_fields_cache', '_clean_fields',
'_clean_form', '_errors', '_html_output', '_post_clean', 'add_error',
'add_initial_prefix', 'add_prefix', 'as_p', 'as_table', 'as_ul', 'auto_id',
'base_fields', 'changed_data', 'clean', 'data', 'declared_fields', 'empty_permitted',
'error_class', 'errors', 'field_order', 'fields', 'files', 'full_clean', 'has_changed',
'has_error', 'hidden_fields', 'initial', 'is_bound', 'is_multipart', 'is_valid',
'label_suffix', 'media', 'non_field_errors', 'order_fields', 'prefix',
'use_required_attribute', 'visible_fields']
```

选择几个后面要用到的关键点来学习一番。

```
>>> login_form.is_bound
False
```

上面的代码说明 login_form 是未绑定的实例。如果我们在创建实例时，向类提供初始化的参数，情况就发生变化了。

```
>>> data_login = {"username":"canglaoshi", "password":"1qaz2wsx"}
>>> lf = LoginForm(data_login)
>>> lf.is_bound
True
```

有两个方法，我们将在随后的视图函数中使用，需要在这里熟悉它们。

```
>>> lf.is_valid()
True
```

在创建实例时，如果传递给表单类的数据是正确的，则 is_valid()返回 True。换个角度说，当前端提交到后端的数据是符合表单类属性要求的，那么 is_valid()返回 True，这个方法用来检验数据是否合法。

```
>>> data_error = {"username":"canglaoshi", "password":''}
>>> error_lf = LoginForm(data_error)
>>> error_lf.is_valid()
False
```

password 是不能为空的（参见前面打印出来的 HTML，没有用专门的参数声明可以为空，默认都是不能为空的），所以上述数据相对表单类不合法，检验结果是 False。

```
>>> lf.cleaned_data
{'password': '1qaz2wsx', 'username': 'canglaoshi'}
```

cleaned_data 是实例的属性，它以字典形式返回实例的具体数据，即经过检验之后的属性及其值。如果传入的某项数据不合法，则在 cleaned_data 的结果中不予显示。

```
>>> error_lf.cleaned_data
{'username': 'canglaoshi'}
```

上述内容搞清楚了，就可以着手在 views.py 中编写函数。

2.2.3　登录的视图函数

还记得如何编写视图函数吗？如果忘记了，可以看看第 1 章中那个简单的视图函数。

此处为响应登录的视图函数，比前面的视图函数稍微复杂一些。我们要解决两个问题，一个是响应 GET 请求，当用户通过浏览器请求某个网址时，显示登录的对话框；另外一个是响应 POST 请求，用户以 POST 方式向服务器发送自己的用户名和密码，该视图函数接收到后，判断其是否为本站用户，如果是就许可登录，否则拒绝。如图 2-7 所示就是登录的视图函数的执行过程。

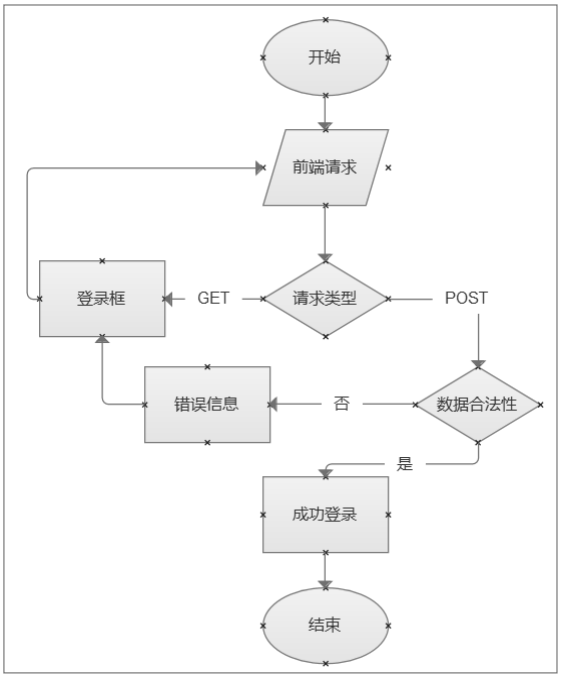

图 2-7　登录流程

根据图 2-7 所表达的逻辑过程，编辑./account/views.py 文件，实现一个名为 user_login()的视图函数。

```
from django.shortcuts import render
from django.http import HttpResponse
from django.contrib.auth import authenticate, login    #①
from .forms import LoginForm

def user_login(request):    #②
    if request.method == "POST":    #③
        login_form = LoginForm(request.POST)    #④
        if login_form.is_valid():    #⑤
            cd = login_form.cleaned_data    #⑥
            user = authenticate(username=cd['username'], password=cd['password'])   #⑦
            if user:
                login(request, user)    #⑧
                return HttpResponse("Wellcome You.
                                    You have been authenticated successfully")    #⑨
            else:
                return HttpResponse("Sorry. Your username or password is not right.")
        else:
```

```
            return HttpResponse("Invalid login")

    if request.method == "GET":
        login_form = LoginForm()
        return render(request, "account/login.html", {"form": login_form})
```

下面需要对此段代码的部分内容进行解释，因为又出现了一些新面孔。

语句①是从 Django 默认的（或者说是内置的）用户认证和管理应用中引入的两个方法。用户管理是一个比较复杂的工作，但是 Django 中内置了一个相当不错的用户认证和管理应用，前面已经通过 admin 账号登录后台管理端对用户进行管理了，此功能就是使用了 Django 内置的用户认证和管理应用。读者要想了解 Django 内置了哪些应用，可以打开./mysite/settings.py 文件，在 INSTALLED_APPS 的值中查看，除自己添加的外，其他都是 Django 内置的，比如语句①中的 django.contrib.auth 就在其中。在这里我们要使用其中两个方法 authenticate()和 login()，从 Python 的角度看，它们可以被视为一个类中的两个方法。

语句②命名了一个视图函数，在这个视图函数中要处理前端提交的数据，并支持前端的显示请求，重要的事情总要多说几遍，视图函数必须使用 request 作为第一个参数。当客户端浏览器通过 URL 向服务器发送请求时，Django 会创建一个 HttpRequest 对象，request 是 HttpRequest 的替代。在 request 对象（或者说 HttpRequest 对象）的诸多方法中，常用的有 request.method、request.GET 和 request.POST。

语句③中的 request.method 就是 HttpRequest 对象的一个常用属性，它返回 HTTP 请求类型的字符串，比如前端向服务器发出的是 GET 请求，那么 request.method 得到“GET”这个字符串。所以，通常像下面这样来使用它。

```
if request.method == 'GET':
    do_something()
if request.method == 'POST':
    do_something_else()
```

request.GET 也是常用属性，虽然上述代码中没有用到，但有必要了解一下。当客户端浏览器向服务器发送 GET 请求后，request.GET 得到一个类字典对象。比如客户端发出 http://www.itdiffer.com/?name=canglaoshi&course=28 的 GET 请求，也就是在 URL 中访问该地址，这个请求其实携带了两个参数及其值，即 name=canglaoshi 和 course=28，两个参数之间用&符号连接。请求发出后，Django 就通过 request.GET 得到两个参数及其值，结果是类字典对象。所以，还可以用 request.GET.get("name")得到参数 name 的值 canglaoshi。请读者注意区分 GET 和 get()，request.GET 得到的是包含所提交请求所有参数和值的类字典对象；get()是字典中的一个方法（详见《跟老齐学 Python：轻松入门》有关字典的章节）。

语句④中用到了 request.POST。在网络上，常用 POST 和 GET 两种方式实现客户端浏览器和服务器之间的数据交互。GET 多用于数据查询，而 POST 多用于数据写入或者更新等（这种划分不是绝对的，根据具体业务要求确定）。在本应用中，前端浏览器向服务器端提交表单内容，采用了 POST 方法。用 POST 方法提交数据，浏览器的地址栏不会发生变化，这与 GET 方法不同。在 Django 中，通过 request.POST 得到提交的表单数据，也是一个类字典对象。

在 Django 中，可以给表单类传入字典类型的数据，从而建立一个绑定实例，在语句④中传入的是 request.POST 返回的类字典数据，效果与前面一样，依然是建立了一个绑定实例。

语句⑤中的 login_form.is_valid()验证所传入的数据是否合法，如果合法就执行后续的语句⑥cd = login_form.cleaned_data，cd 引用的是一个字典类型数据，其中以键值对的形式记录了用户名和密码。

语句⑦中用到了语句①所引入的 authenticate()函数，其作用是检验此用户是否为本网站项目的用户，以及其密码是否正确。如果都对上号了，就返回 User 的一个实例对象，否则返回 None。在交互模式下可以尝试一番。

```
>>> from django.contrib.auth import authenticate, login
>>> user = authenticate(username="zhanglaoshi", password="1qazxcvbnm")
#前提，我已经通过 admin 增加了 zhanglaoshi 这个账号
>>> user
<User: zhanglaoshi>
>>> user = authenticate(username="zhanglaoshi", password="12345678")
>>> user
>>>
```

语句⑧中使用了语句①引入的 login()函数，以语句⑦所得到的 User 实例对象作为参数，实现用户登录。用户登录之后，Django 会自动调用默认的 session 应用，将用户 ID 保存在 session 中，完成用户登录操作（关于 session 的有关知识，请读者自行搜索资料了解）。在通常情况下，login()和 authenticate()配合使用。

如果将网页中的 HTML 看作一个字符串，就可以传给 HttpResponse 类（这个类在 django.http 模块内，所以前面要使用 from django.http import HttpResponse），从而在客户端返回一个页面，这就是语句⑨的含义，这一点和 render 有所不同。

如果客户端对服务器发出的是 GET 请求，则将 login_form = LoginForm()传给指定模板，从而在前端呈现相应的对话框。

编写完登录的视图函数，为了能够在前端显示登录对话框，还要编写相应的前端模板。

2.2.4 登录的前端界面

接下来要编写./templates/account/login.html 模板（建立 account 子目录），代码如下。

```
{% extends "base.html" %}

{% block title %}Login{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Login</h1>
    <p>Input your username and password</p>
    <form class="form-horizontal" action="." method="post">{% csrf_token %}    #①
        {{ form.as_p}}    #②
        <input type="submit" value="Login">
    </form>
</div>
{% endblock %}
```

模板编写好之后，总有一种迫不及待看效果的冲动。在查看效果之前，请读者确认 Django

服务已重新启动，本书后面也会常常提醒，因为很多初学者容易忘记这个重要但简单的操作。如果增加了新的文件，一般要重新启动 Django 服务。在浏览器的地址栏中输入 http://localhost:8000/account/login，就能看到漂亮的登录界面了，如图 2-8 所示。

图 2-8　登录界面

为了此处测试的需要，笔者已经在前面利用超级管理员创建了用户，这里就使用该用户登录，比如 zhanglaoshi，密码是 1qazxcvbnm，单击 Login 按钮，就会返回一个消息界面，如图 2-9 所示。

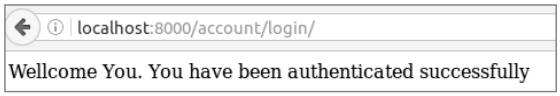

图 2-9　登录成功

相当成功的登录，至少目前觉得是。不要激动，还要慢慢对其中的代码进行深度了解。

语句①中的<form>是创建一个前端显示的表单，其中的 action="."用于显示当前表单的提交地址，这里表示的是当前地址，即调用此模板的视图函数所对应的 URL。

语句①中的{% csrf_token %}在表单中是必不可少的，不写在这个位置也行，只要在<form>语句里面即可。为什么说它不可缺少呢？这要从 CSRF 开始说起，所谓 CSRF（Cross-Site Request Forgery），中文名称是跨站请求伪造，也被称为 one click attack/session riding，简而言之就是攻击者盗用了用户的身份，以用户的名义发送恶意请求，导致的不良后果读者可以想象一下。Django 作为一个负责任的框架，内置了一个 CSRF 中间件，请读者打开./mysite/settings.py 文件，虽然前面在这里进行两次配置应用了，但对于 MIDDLEWARE 读者或许没有关注，现在就看看它，其中有一个参数 django.middleware.csrf.CsrfViewMiddleware，有了这个中间件，就能保证 Django 免受 CSRF 攻击了。如果前端通过 POST 方式提交数据，就会被禁止，而 POST 数据又是必不可少的，解决方法之一就是在表单中使用{% csrf_token %}，与表单内容一同被提交。为了验证，读者不妨将其暂时删除，看看是否会出现如图 2-10 所示的结果。

图 2-10　禁止 CSRF

前面曾经打印了一个未绑定表单实例对象，结果是 HTML 的表单框（如果忘记了，请马上复习前面的内容，不要匆匆而过）。当请求为 GET 时，视图函数中产生的就是未绑定对象，用这个对象来渲染前端模板，得到一个等待输入的表单，这就是语句②，只不过是使用了实例对象的 as_p 方法，它使得表单显示为一系列<p>标签，每个表单元素在一对<p>标签内。与之类似的还有 as_ul 和 as_table。如果查看此时的 http://localhost:8000/account/login/页面源码，就会发现如图 2-11 所示的<p>标签所包裹的元素。

图 2-11　CSRF 的效果

读者可能注意到，页面的导航条中有一个预设的 LOGIN，能不能在这里做一个超级链接，指向现在做的登录页面呢？

当然可以。不过要编辑./templates/header.html 中的部分代码，打开这个文件，新修改的部分代码如下。

```
{% load staticfiles %}
<div class="container">
    <nav class="navbar navbar-default" role="navigation">
        ……
        <div>
            ……
            <ul class="nav navbar-nav navbar-right" style="margin-right:10px">
                <li><a href="{% url 'account:user_login' %}">LOGIN</a></li>    #③
            </ul>
#省略余下代码
```

语句③做了一个超链接<a href="{% url 'account:user_login' %}">，注意这里需要在项目和应用的 urls.py 中做相应的配置，请参考 2.1.3 节中的描述（本节在前面的相关代码已经考虑到此处需要的配置）。

为了查看效果，先访问 http://localhost:8000/blog/，因为在模板中使用的都是 head.html，所以导航条右边同样有 LOGIN，如图 2-12 所示。

图 2-12　从博客页面转到登录页面

单击 LOGIN 按钮，就会跳转到刚刚创建的用户登录页面。

可能感觉很不错了，其实还有另外的实现方法。在这里我们使用{{ form.as_p }}的方式得到了前端模板所要渲染的表单，但是这种方式略显死板。比如，要求 Username 用红色字而 Password 用蓝色字，上面写法就不能实现了。于是就需要对代码进行修改，以下为修改./templates/account/login.html 文件后的代码。

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}Login{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Login</h1>
    <p>Input your username and password</p>
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <!-- {{ form.as_p}}-->
        <div class="form-group">
            <label for="{{ form.username.id_for_label }}"
                   class="col-md-5 control-label" style="color:red">
               <span class="glyphicon glyphicon-user"></span>Username</label>
            <div class="col-md-6 text-left">{{ form.username }}</div>
        </div>
        <div class="form-group">
            <label for="{{ form.password.id_for_label }}"
                   class="col-md-5 control-label" style="color:blue">
               <span class="glyphicon glyphicon-floppy-open"></span>Password</label>
            <div class="col-md-6 text-left">{{ form.password }}</div>
        </div>
        <input type="submit" class="btn btn-primary btn-lg" value="Login">
    </form>
</div>
{% endblock %}
```

以上貌似很多代码，其实并没有什么深奥的，只是更多地使用了 Bootstrap 的样式罢了。关键是里面的{{ form.username}}和{{ form.password }}，用这种方式单独实现了<input>元素。页面的部分展示效果如图 2-13 所示。

图 2-13　新登录界面

如果读者是跟随本书中的进度在调试，那么就会在浏览器中看到一个色彩比较鲜艳的界面。对于这个新的登录页面，请读者自行测试。

从本书开始到现在，读者是否已经体会到了用 Django 做项目的乐趣？做个快乐的程序员吧！程序员的快乐就来自指尖流淌着的代码，将想象变成眼前的现实。

2.2.5 知识点

1．内置用户权限管理

利用 Django 开发网站，可以使用其内置的用户权限管理功能，也可以自己重新编写一个用户管理的应用。在本书所展示的项目中，都是使用 Django 默认的用户管理应用。如果读者是跟着本书的流程来实践的，那么打开./mysite/settings.py 文件，会在 INSTALLED_APPS 的列表中发现 django.contrib.auth 应用，这是创建本项目时默认就有的，它就是 Django 默认（内置）的用户权限管理应用。这个应用的所有源码都存放在本机上的 Django 安装目录的./django/contrib/auth 内。

在完成数据迁移后，可以在数据库中看到以 auth 作为前缀的数据库表，其中 auth_user 是我们在本节中要关注的，用如下方式查看它的结构。

```
sqlite> pragma table_info(auth_user);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           password    varchar(12  1                       0
2           last_login  datetime    0                       0
3           is_superus  bool        1                       0
4           username    varchar(15  1                       0
5           first_name  varchar(30  1                       0
6           email       varchar(25  1                       0
7           is_staff    bool        1                       0
8           is_active   bool        1                       0
9           date_joine  datetime    1                       0
10          last_name   varchar(15  1                       0
```

到现在，如果读者对 SQL 语句还不熟悉，建议你尽快学习相关知识。

完成用户的登录或者 2.4 节中的注册，就是针对这个表进行的操作，它对应着./contrib/auth/models.py 文件中的数据模型类 User。

用超级管理员身份登录（http://localhost:8000/admin），在页面找到对用户进行管理的地方，查看当前用户，并且能够对用户进行增加、删除、完善信息等操作。这些都是 Django 默认的 auth 应用中所具有的功能。

Django 默认的用户管理应用功能非常强大，本节仅使用了登录和退出两个简单的功能。

读者如果有兴趣全面了解，可以阅读官方文档（https://docs.djangoproject.com/en/2.1/topics/auth/default/，要阅读英文的文档，不推荐阅读翻译的文档，因为编程是一项严谨的工作，不能有任何差错，而翻译的文档跟原文相比难免有出入，即使很小的差别也可能导致读者理解上的偏差，造成编程的错误），不管现在是否能够全面理解，把其内容阅读一遍，留有印象，很可能在将来某个时候会用到。

2．表单的请求响应过程

在 Web 上，表单的用途非常多。Django 是如何对响应前端发送表单请求的呢？如图 2-14 所示显示了这个过程。

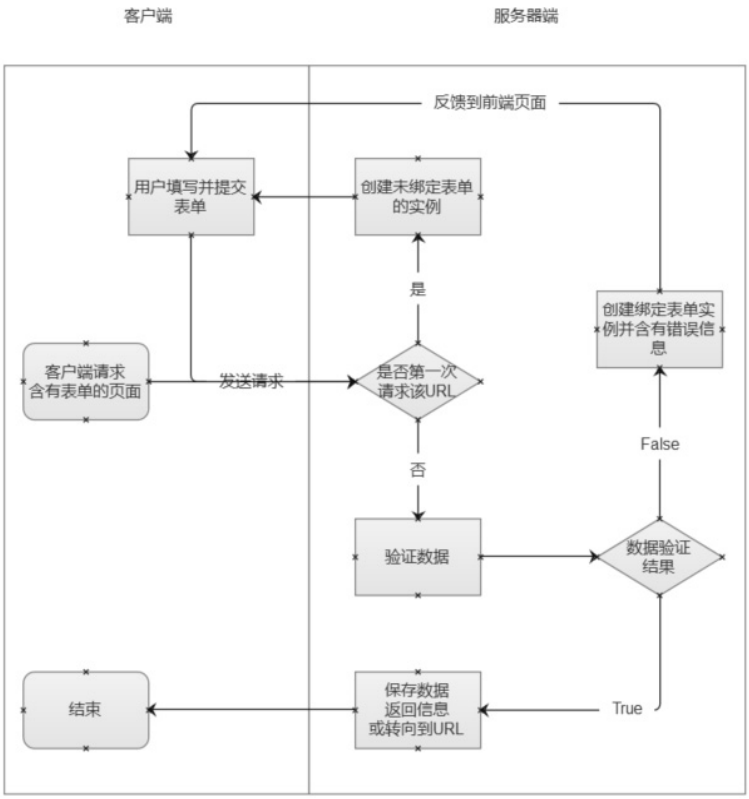

图 2-14　表单响应过程

（1）用户通过自己的浏览器（客户端）第一次向服务器发出含有表单页面的请求，Django 会创建一个未绑定数据的表单实例（例如 form = LoginForm()，form 实例就是未绑定实例），然后反馈到前端页面，等待用户填写内容。

（2）用户在客户端填写了表单内容之后，将其提交给服务器，在 Django 的视图中接收数据（例如 form = LoginForm(request.POST)），然后验证表单数据（form.is_valid()）。

（3）通过表单验证之后，可以对表单进行进一步操作，例如保存、URL 转向等。结束之后，本次表单提交过程完毕。

（4）如果没有通过表单验证，就要返回绑定表单实例（携带已经提交的数据和错误信息），让用户修改之后再次提交。

3．文档导读

（1）*Using the Django authentication system*，https://docs.djangoproject.com/en/2.1/topics/auth/default/。

（2）*User Authentication in Django*，http://djangobook.com/user-authentication-django/。

2.3 用内置方法实现登录和退出

前面花费了不少时间学习用户登录功能，对于网站而言登录是非常普遍的，而且功能雷同，都是输入用户名和密码。在 Django 中，凡是通用的功能，都尽可能提供了默认的方法，不需要用户编写代码，只需调用即可实现。

下面学习 Django 内置的登录方法。在使用之前，要找到这个方法写在什么地方了，并且要看看源码。用内置方法实现登录和退出的基本知识如图 2-15 所示。

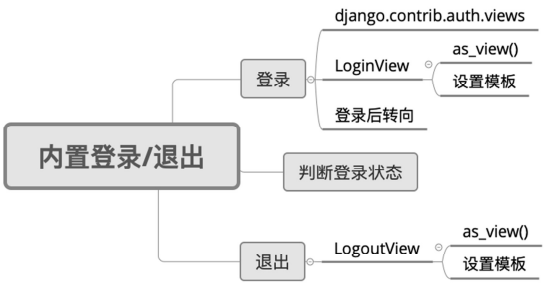

图 2-15 知识技能导图

2.3.1 内置的登录方法

在 2.2 节中，我们已经亲自动手写了一个用于登录的视图函数（./account/views.py 中的 user_login 函数）。本来，那个视图函数已经实现了登录功能，但是，本着“深入理解、必须不断折腾”的理念，本节将演示如何使用 Django 内置的方式实现登录——不需要自己动手写登录的视图函数，以便更深入地掌握 Django 开发的快捷方法。

首先声明，读者可能会对以下写法产生疑惑，但是，请暂且“不求甚解”，先按照操作步骤实现功能，经过耐心思考及操作，自然会豁然开朗。

在./account/urls.py 中增加新的引入。

```
from django.contrib.auth import views as auth_views
```

然后按照下面的方式修改 urlpatterns 的值。

```
urlpatterns = [
    #path('login/', views.user_login, name='user_login'),
    path('login/', auth_views.LoginView.as_view(template_name='account/login2.html'),
name='user_login'),
]
```

把 2.2 节中写的登录路径注释掉，增加新的登录路径配置。其中 auth_views.LoginView.as_view()是新引入的模块中提供的登录方法，这就是本节所说的 Django 内置登录方法。只不过，这里是以“基于类的方式”写的——我们前面一直是以“基于函数”的方式实现视图文件中的功能——关于“基于类”视图的详细阐述，请阅读本书第 7 章。

auth_views.LoginView.as_view()中的参数 template_name 声明了支持此登录的前端页面。创建./templates/account/login2.html，其完整代码如下（注意与 2.2 节中所写的./account/login.html 不同）。

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}Login{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Login--Built in Method of Django</h1>

    {% if form.errors %}    #①
        <p>Your username and password didn't match. Please try again.</p>
    {% endif %}

    <p>Input your username and password</p>
    <form class="form-horizontal" action="{% url 'account:user_login' %}" method="post">
{% csrf_token %}    #②
        <div class="form-group">
            <label for="{{ form.username.id_for_label }}" class="col-md-5 control-label"
style="color:red">
                <span  class="glyphicon  glyphicon-user"></span>Username</label>  <div
class="col-md-6 text-left">{{ form.username }}</div>
        </div>
        <div class="form-group">
            <label for="{{ form.password.id_for_label }}" class="col-md-5 control-label"
style="color:blue">
                <span class="glyphicon glyphicon-floppy-open"></span>Password</label>
            <div class="col-md-6 text-left">{{ form.password }}
            </div>
        </div>
        <input type="submit" class="btn btn-primary btn-lg" value="Login">
    </form>
</div>
{% endblock %}
```

相比./templates/account/login.html，上述文件中的语句①是新增的，其功能在于，如果登录信息有误，则会显示<p>标签内的提示信息。

语句②相对原来也有所变化，以 action="{% url 'account:user_login' %}"的方式明确了表单中数据的 POST 目标。

重新启动 Django 服务，在浏览器的地址栏中输入 http://localhost:8000/account/login/，可以看到登录对话框，与前面一样，输入用户名和密码，单击 LOGIN 按钮，结果如图 2-16 所示。

并没有如愿以偿地出现那个登录成功的界面，而是出现 404 错误。

对于初学者而言，出现各种各样的错误是很正常的，虽然面对错误会有挫败感。这时候特别需要给自己打打气，“失败是成功之母”!

现在凝神定气、仔细观察，发现请求的地址是 http://localhost:8000/accounts/profile/，这是我们从来没有配置过的地址。或者说，在我们提交登录之后，自动转向到了这个地址。是什么力量让访问转向了呢？

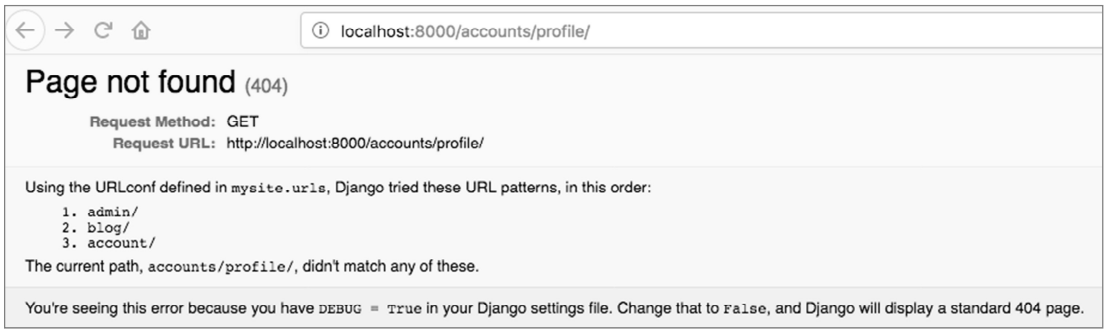
localhost:8000/accounts/profile/
Page not found (404)
Request Method: GET
Request URL: http://localhost:8000/accounts/profile/
Using the URLconf defined in mysite.urls, Django tried these URL patterns, in this order:
1. admin/
2. blog/
3. account/
The current path, accounts/profile/, didn't match any of these.
You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.

图 2-16　登录后报错界面

在 Django 中，默认配置了登录之后页面的转向地址，在官方文档中对此有明确说明（https://docs.djangoproject.com/en/2.1/ref/settings/#std:setting-LOGIN_REDIRECT_URL），即 LOGIN_REDIRECT_URL 的默认值是'/accounts/profile/'。为了设置项目所需要的转向目标，需要在./mysite/settings.py 中增加 LOGIN_REDIRECT_URL 的值（将以下配置放置到./mysite/settings.py 文件的最后）。

```
LOGIN_REDIRECT_URL = '/blog/'
```

这行代码的含义是登录后重定向到 http://localhost:8000/blog/页面，读者当然可以设定为任何一个可以访问的地址作为重定向目标地址。

再次登录，发现登录成功之后神奇地转到了指定页面。

2.3.2　判断用户是否登录

在模板中，常常需要根据用户是否登录进行不同的显示。现在就来修改./templates/header.html 模板，使 LOGIN 所在的位置根据用户登录状态显示不同的内容。如果用户已经登录，则显示用户名和 Logout 字样；如果用户尚未登录，则显示 LOGIN 字样，并做超链接到登录界面。

```
{% load staticfiles %}
<div class="container">
    <nav class="navbar navbar-default" role="navigation">
        <div class="navbar-header">
            <a class="navbar-brand" href="http://www.itdiffer.com">
               <img src="{% static '/images/logo.png' %}" width="100px">
            </a>
        </div>
        <div>
            <ul class="nav navbar-nav" role="navigation">
                <li><a href="{% url 'blog:blog_title' %}">BLOG</a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right" style="margin-right:10px">
                {% if user.is_authenticated %}    #①
                <li><a href="#">{{ user.username }}</a></li>
                <li><a href="#">Logout</a></li>
                {% else %}
                <li><a href="{% url 'account:user_login' %}">LOGIN</a></li>
                {% endif %}
```

```
            </ul>
        </div>
    </nav>
</div>
```

在模板中，语句①类似 Python 中的 if 语句，请读者注意观察这种模板语言的书写规则。在模板中可以通过 request.user 得到用户对象，user.is_authenticated 则是返回用户登录的状态，已经登录返回 True，否则返回 False。这样在模板中就可以根据用户的状态显示不同的内容。

上述代码中 Logout 和用户名的超链接都留空了，后续会填补上。登录之后的导航条状态如图 2-17 所示。

图 2-17　登录状态

显示了登录用户的用户名和 Logout。所谓 Logout，即为退出。有进有出，才合情理。

2.3.3　内置的退出方法

Django 不仅有内置的 LoginView 可以实现登录，还有实现退出功能的内置类，因为退出也是所有网站必备的，故 Django 把这个功能内置了。

与登录类似，首先配置 URL。

编辑./account/urls.py 文件，增加如下内容。

```
urlpatterns = [
    …
    path('logout/', auth_views.LogoutView.as_view(), name='user_logout'),
]
```

此处配置好后，把./templates/header.html 文件中那个针对 Logout 的超链接修改一下。

```
<li><a href="{% url 'account:user_logout' %}">Logout</a></li>
```

检查 Django 服务是否在运行，刷新页面。如果没有登录，则找一个用户名来登录。一切准备就绪，单击 Logout 按钮，该用户退出，并且打开如图 2-18 所示的页面。

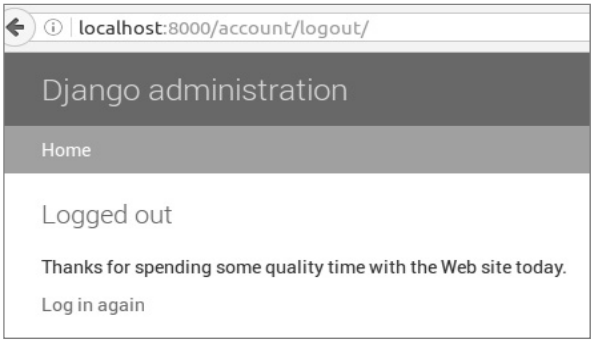

图 2-18　退出后的界面

这个页面是 Django 为退出函数定义好的界面，虽然很丑，但毕竟实现了退出。

的确是丑了点，因为使用的是 Django 的默认模板，下面我们要在./templates/account/中创建一个退出的模板文件，名为 logout.html，代码如下。

```
{% extends "base.html" %}

{% block title %}Logout{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <p>You have log out.</p>
    <p>You can <a href="{% url 'account:user_login'%}">login</a> again</p>
</div>
{% endblock %}
```

仿照前面登录页面的 URL 配置，修改退出的 URL 配置（./account/usrls.py 文件）。

```
path('logout/', auth_views.LogoutView.as_view(template_name='account/logout.html'),
     name='user_logout'),
```

再从登录状态执行退出操作，效果如图 2-19 所示。

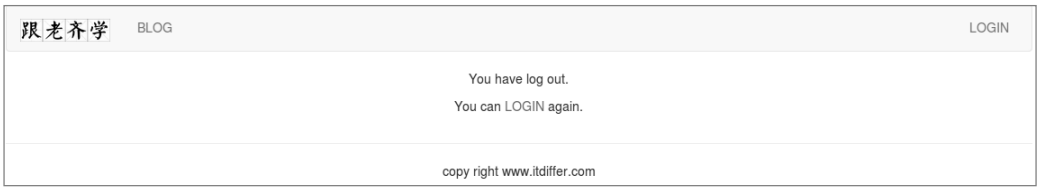

图 2-19　自定义退出后的界面

注意观察导航条的右边，这时候显示的是 LOGIN 了。

正确地使用内置方法，能够让开发更快速——我们不生产代码，我们是代码的搬运工。

2.3.4　知识点

Django uses request and response objects to pass state through the system.

这句话源自 https://docs.djangoproject.com/en/2.1/ref/request-response/，请读者认真阅读。

1．HttpRequest 对象

当客户端向服务器发送请求时，Django 会创建一个包含请求数据的 HttpRequest 对象，并以参数 request 传给视图函数，即参数 request 所引用的对象就是 HttpRequest 对象。HttpRequest 对象有很多属性，下面列出几项，供读者查阅参考（除 session 外的其他属性都应该被看作是只读的，并且下列各项属性区分大小写）。

- path：返回一个字符串，表示请求页面的路径（不包括域名），例如“/blog/admin/”。
- GET：返回类字典对象，包括所有的 HTTP GET 参数。
- POST：返回类字典对象，包括所有的 HTTP POST 参数。注意，不包括上传文件信息。
- REQUEST：返回类字典对象，包括所有 POST 和 GET 数据（先 POST，后 GET；在项目实践中，不推荐使用这个属性获得对象数据，使用 GET、POST 更清晰）。

- FILES：返回类字典对象，包含所有的上传文件，每个键是<input type="file" name="" />中 name 的值，每个值是一个 Python 字典，该字典有以下三个键：①filename，上传文件的文件名；② content-type，上传文件的内容类型；③ content，上传文件的原始内容。注意 FILES 只有在请求方式为 POST 并且表单包括 enctype="multipart/ form-data" 属性时才会有数据，否则 FILES 就是一个空的类似字典的对象。
- COOKIES：返回字典，包括所有的 cookie 键值对。
- META：返回字典，包含所有可能的 HTTP 头。
- user：返回 django.contrib.auth.users.User 对象，表示当前登录用户。如果当前没有用户登录，user 被设置成 django.contrib.auth.models.AnonymousUser 的一个实例，可以用 is_authenticated（或 is_anonymous）属性来区分登录用户和未登录用户，例如：

```
if request.user.is_authenticated:
    # Do something for anonymous users.
else:
    # Do something for logged-in users.
```

- session：返回可读写的类字典对象，表示当前的 session。

除上述属性外，HttpRequest 对象还有一些方法，读者可以到官方文档中查看。

在一个 HttpRequest 对象中，GET 和 POST 属性都是 django.http.QueryDict 的实例。QueryDict 是一个类字典的对象，可以处理同一个键有多个值的情况。

2．HttpResponse 对象

对应着 HttpRequest 对象，HttpResponse 是视图向客户端返回的对象。典型的用法就是将页面的内容作为字符串传递给 HpptResponse 函数。例如（以下示例来自官方文档）：

```
>>> from django.http import HttpResponse
>>> response = HttpResponse("Here's the text of the Web page.")
>>> response = HttpResponse("Text only, please.", content_type="text/plain")
```

如果需要随时增加内容，还可以像操作文件一样来操作 HttpResponse 对象。

```
>>> response = HttpResponse()
>>> response.write("<p>Here's the text of the Web page.</p>")
>>> response.write("<p>Here's another paragraph.</p>")
```

HttpResponse 对象也有一些属性和方法，在这里就省略了，请读者认真阅读前面所提到的官方文档，详细了解相关属性和方法，此外，这个对象还有一些子类，在某些情况下也会用到，请读者抽时间自行浏览。

3．文档导读

（1）*Request and response objects*，https://docs.djangoproject.com/en/2.1/ref/request-response/。

（2）*How to Work With AJAX Request With Django*，https://simpleisbetterthancomplex.com/tutorial/2016/08/29/how-to-work-with-ajax-request-with-django.html。

2.4 用户注册

此前所使用的账户是通过超级管理员设置的，这种方式通常被某些比较封闭的网站采用，比如某机构的内部管理网站，目的是要严格审查用户身份。对于社会性网站，一般允许通过自行注册的方式成为网站用户。

所以，要学习如何实现注册功能，如图 2-20 所示。

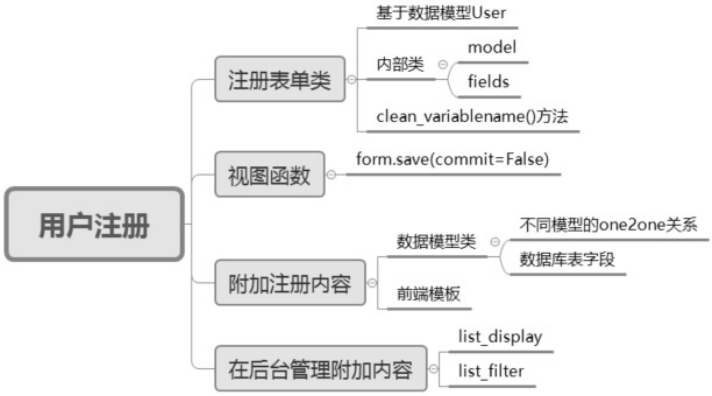

图 2-20　知识技能导图

2.4.1 简单注册

不能一口吃个胖子，先看一个简单的例子。

最简单的就是注册用户名和密码，并且也不单独创建数据库，利用原有的数据库，即 Django 默认的用户数据模型。

实现用户注册功能与实现登录功能类似，都要填写表单。所以，编写表单类是必需的。编辑./account/forms.py 文件，增加一个注册用户的表单类。

```
from django import forms
from django.contrib.auth.models import User    #①
…
class RegistrationForm(forms.ModelForm):    #②
    password = forms.CharField(label="Password", widget=forms.PasswordInput)
    password2 = forms.CharField(label="Confirm Pssword", widget=forms.PasswordInput)

    class Meta:    #③
        model = User    #④
        fields = ("username", "email")

    def clean_password2(self):    #⑤
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError("passwords do not match.")
        return cd['password2']
```

这个表单类比前面定义的 LoginForm 类复杂，并且有一些差别。

语句①引入了 Django 默认的用户模型 User 类，在这个表单类中就应用 User 模型，不需要再新建用户数据模型了。

前面的登录表单类中曾经继承了 forms.Form 类，语句②中的 forms.ModelForm 是 forms 里面的另外一个常用的类，这两个类在应用上是有区别的。一般情况下，如果要将表单中的数据写入数据库表或者修改某些记录的值，就要让表单类继承 ModelForm 类；如果提交表单之后，不会对数据库进行修改，则继承 Form 类。

语句③是一个内部类，其中我们需要声明本表单类所应用的数据模型，也就是将来表单的内容会写入哪个数据库表中的哪些记录里面。表单类中的属性和数据模型类中的属性一一对应，有时候我们不必在表单中向数据模型中的所有字段赋值（或者不需要把数据库表中所有的字段写入数据），可以用 fields = ("username", "email")来说明这里所选用的属性（数据库表中的字段），或者用 exclude 列表说明所排除的属性。

类 RegistrationForm 的 password 和 password2 两个密码属性是为了让用户确认自己所设置的密码，这里定义的属性意味着覆盖或者不使用在内部类 Meta 的声明数据模型中的字段，在表单中相关字段对应此处重新定义的两个变量。

随后，在类内部还定义了语句⑤中的方法 clean_password2()，这个方法的作用是检验用户所输入的两个密码是否一致。检查的动作在我们调用表单实例对象的 is_valid()方法时会被执行。以“clean_+属性名称”命名方式所创建的方法，都有类似的功能。

数据模型类（User）和表单类（RegistrationForm）都已经有了，接下来就是视图函数和相应的前端模板。

继续编辑./account/views.py 文件，编写注册的视图函数。

```
from .forms import LoginForm, RegistrationForm

def register(request):
    if request.method == "POST":
        user_form = RegistrationForm(request.POST)
        if user_form.is_valid():
            new_user = user_form.save(commit=False)    #①
            new_user.set_password(user_form.cleaned_data['password'])    #②
            new_user.save()
            return HttpResponse("successfully")
        else:
            return HttpResponse("sorry, your can not register.")
    else:
        user_form = RegistrationForm()
        return render(request, "account/register.html", {"form": user_form})
```

这个视图函数和 2.2 节编写的 user_login()函数的基本结构相似，只是在对表单的处理上稍有差别。

ModelForm 类或者它的子类都具有 save()方法，它的效果是将表单数据保存到数据库，并生成该数据对象。但是，语句①应用此方法时，在参数里多了 commit=False，其结果是数据并没有被保存到数据库，而仅生成了一个数据对象。紧接着看后面两句，语句②设置了该数据对

象的密码（这个密码是经过校验的），这个操作得益于前面 commit=False 的设置，如果没有此参数，数据被写到数据库中了，再设置密码就需要重新写入，稍显浪费且慢了。

密码已经保存到数据对象了，再执行 new_user.save()就可以把数据保存到数据库中。

前端模板与登录模板类似，就不多解释了，在.templates/account 里面建立一个名为 register.html 的文件，注意不是在 registration 目录中，因为这个注册功能完全是我们自己写的，代码如下。

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}register user{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Register</h1>
    <p>If you are a user, <strong><a href="{% url 'account:user_login' %}">Login</a>
</strong>,please</p>
    <p>or register.</p>
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="form-group">
            <label for="{{ form.username.id_for_label }}" class="col-md-5 control-label">
Username</label>
            <div class="col-md-6 text-left">{{ form.username }}</div>
        </div>
        <div class="form-group">
            <label for="{{ form.email.id_for_label }}" class="col-md-5 control-label">
Email</label>
            <div class="col-md-6 text-left">{{ form.email }}</div>
        </div>

        <div class="form-group">
            <label for="{{ form.password.id_for_label }}" class="col-md-5 control-label">
Password</label>
            <div class="col-md-6 text-left">{{ form.password }}</div>
        </div>
        <div class="form-group">
            <label for="{{ form.password.id_for_label }}" class="col-md-5 control-label">
Confirm Password</label>
            <div class="col-md-6 text-left">{{ form.password2 }}</div>
        </div>
        <input type="submit" class="btn btn-primary btn-lg" value="REGISTER">
    </form>
</div>
{% endblock %}
```

最后不要忘记了，还有./account/urls.py 中的配置，代码如下。

```
urlpatterns = [
    …
    path('register/', views.register, name='user_register'),
]
```

检查 Django 服务是否重新启动了，然后打开浏览器，访问 http://localhost:8000/account/register，打开如图 2-21 所示的页面。

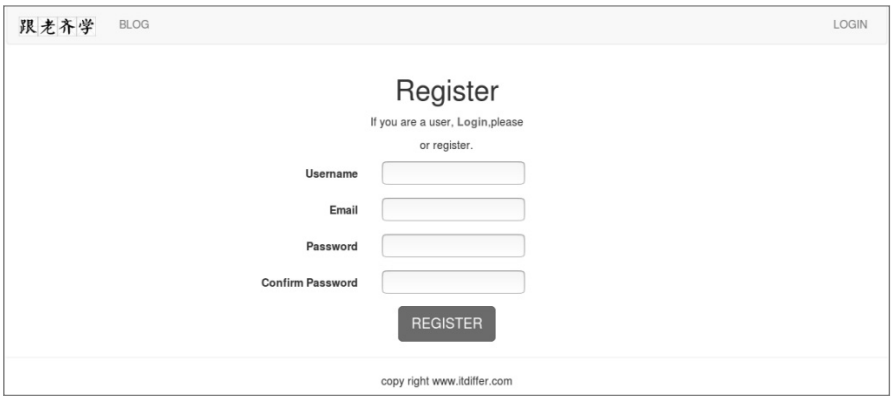

图 2-21　简单的注册界面

输入用户名和密码进行注册，如果能够看到如图 2-22 所示的结果，就说明注册成功了。

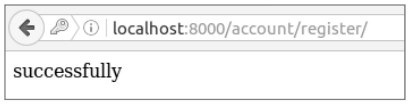

图 2-22　注册成功

这是根据视图函数 register()的规定返回的结果。

虽然上述注册信息比较简单，但是读者了解了注册的流程，稍微复杂的实例在后面。

2.4.2　增加注册内容

对于注册的过程，不同的网站有不同的需要，所以 Django 并没有像登录、退出那样设置一个内置的方法，这就给开发者留下了很多发挥的空间，比如针对前面的注册过程，如果觉得仅让用户在注册时填写用户名、邮箱和密码有点简单，那么还可增加其他的项目，在默认的 User 数据模型中还有用户姓氏、名字等。如果要增加，可以在表单类的内部类 fields 的值中继续增加属性（数据库表的字段）名称。

但是，要增加一个 User 数据模型中没有的字段（比如手机号）该怎么办？下面就解决这个问题。

为了满足这个需求，首先要有数据模型。于是，编辑./account 中的 models.py 文件，并在其中输入如下代码。

```
from django.db import models
from django.contrib.auth.models import User

class UserProfile(models.Model):    #①
    user = models.OneToOneField(User, on_delete=models.CASCADE, unique=True)   #②
    birth = models.DateField(blank=True, null=True)
    phone = models.CharField(max_length=20, null=True)
```

```
    def __str__(self):
        return 'user {}'.format(self.user.username)
```

语句①所定义的数据模型类对应着要在数据库中建立的名为 account_userprofile 的数据库表。

语句②中定义了名为 user 的字段，OneToOneField()的含义是通过 user 这个字段（作为纽带）声明 UserProfile 类与 User 类之间的关系是“一对一”的。

在项目根目录下依次执行如下命令，迁移数据。

```
$ python3 manage.py makemigrations account
Migrations for 'account':
  account/migrations/0001_initial.py
    - Create model UserProfile

$ python3 manage.py migrate account
Operations to perform:
  Apply all migrations: account
Running migrations:
  Applying account.0001_initial... OK
```

读者计算机的显示结果可能与笔者计算机的显示结果有所差异，但不必为此惊讶，重点看能不能实现最终界面的显示效果。

可以用下面的方式查看已有数据库表，会看到刚刚同步的数据库表。

```
sqlite> .tables
account_userprofile        auth_user_user_permissions
auth_group                 blog_blogarticles
auth_group_permissions     django_admin_log
auth_permission            django_content_type
auth_user                  django_migrations
auth_user_groups           django_session
```

的确生成了一个名为 account_userprofile 的数据库表，并且它的表结构与刚才设计的 UserProfile 类中的属性一致。

```
sqlite> pragma table_info(account_userprofile);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           birth       date        0                       0
2           phone       varchar(20) 0                       0
3           user_id     integer     1                       0
```

按照流程，下面应该编写表单类了，编辑./account/forms.py 文件，代码如下。

```
from django import forms
from django.contrib.auth.models import User
from .models import UserProfile
…
class UserProfileForm(forms.ModelForm):
    class Meta:
        model = UserProfile
        fields = ("phone", "birth")
```

这个表单类非常简单，也没有什么好解释的了。下面要完善视图函数。

编辑./account/views.py 文件，增加和修改如下代码。

```
…
from .forms import LoginForm, RegistrationForm, UserProfileForm
…
def register(request):
    if request.method == "POST":
        user_form = RegistrationForm(request.POST)
        userprofile_form = UserProfileForm(request.POST)
        if user_form.is_valid()*userprofile_form.is_valid():
            new_user = user_form.save(commit=False)
            new_user.set_password(user_form.cleaned_data['password'])
            new_user.save()
            new_profile = userprofile_form.save(commit=False)
            new_profile.user = new_user
            new_profile.save()
            return HttpResponse("successfully")
        else:
            return HttpResponse("sorry, your can not register.")
    else:
        user_form = RegistrationForm()
        userprofile_form = UserProfileForm()
        return render(request, "account/register.html",
                      {"form": user_form, "profile":userprofile_form})
```

修改后的视图函数与原来相比，主要是兼顾了另外一个表单类，所以只是在代码行数上增加了，并没有增加新的内容。下面修改模板，重新编辑./templates/account/register.html 文件，代码如下。

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}register user{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Register</h1>
    <p>If you are a user, <strong><a href="{% url 'account:user_login' %}">Login</a>
</strong>,please</p>
    <p>or register.</p>
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="form-group">
            <label for="{{ form.username.id_for_label }}" class="col-md-5 control-label">
Username</label>
            <div class="col-md-6 text-left">{{ form.username }}</div>
        </div>
        <div class="form-group">
            <label for="{{ form.password.id_for_label }}" class="col-md-5 control-label">
Password</label>
            <div class="col-md-6 text-left">{{ form.password }}</div>
```

```
        </div>
        <div class="form-group">
            <label for="{{ form.password.id_for_label }}" class="col-md-5 control-label">
Confirm Password</label>
            <div class="col-md-6 text-left">{{ form.password2 }}</div>
        </div>
        <div class="form-group">
            <label for="{{ profile.birth.id_for_label }}" class="col-md-5 control-label">
Birth Date</label>
            <div class="col-md-6 text-left">{{ profile.birth }}</div>
        </div>
        <div class="form-group">
            <label for="{{ profile.phone.id_for_label }}" class="col-md-5 control-label">
Phone</label>
            <div class="col-md-6 text-left">{{ profile.phone }}</div>
        </div>
        <input type="submit" class="btn btn-primary btn-lg" value="REGISTER">
    </form>
</div>
{% endblock %}
```

只是多了几行代码，没有什么太新奇的。

检查一下 Django 服务是否运行了，然后通过浏览器访问 http://localhost:8000/account/register，效果如图 2-23 所示。

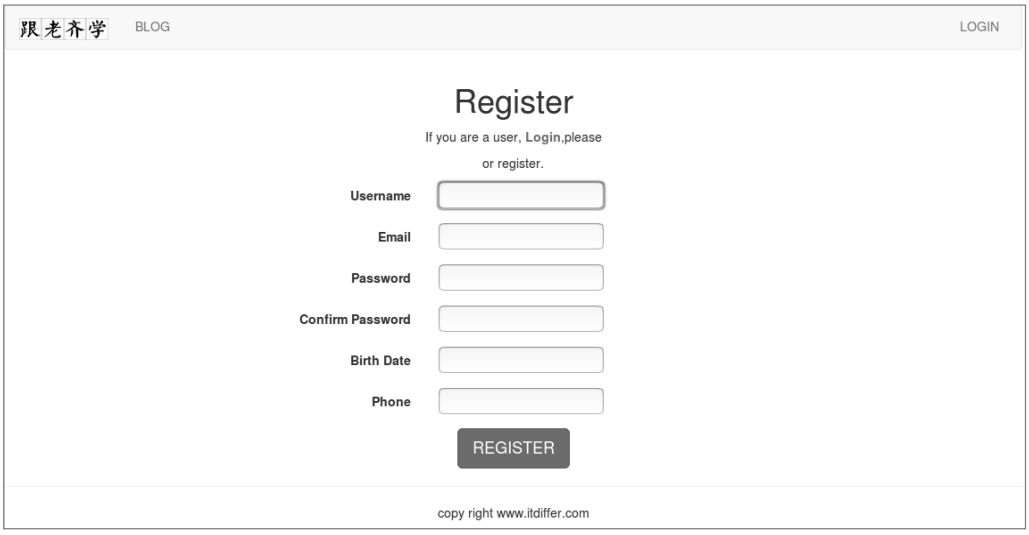

图 2-23　修订后的注册页面

依次填写各项内容后单击 REGISTER 按钮，完成用户注册过程。

如果成功注册了，那么用户信息将被保存到数据库中。查看一下数据库表，会看到新注册的用户信息。

```
sqlite> select id, username, email from auth_user;
id          username    email
----------  ----------  ---------------
```

```
1          admin       admin@admin.com
2          wanglaoshi
3          zhagnlaosh
4          canglaoshi  cang@laoshi.com
5          linlaoshi

sqlite> select * from account_userprofile;
id         birth       phone       user_id
----------  ----------  ----------  ----------
1          1989-06-01  1234567890  5
```

其中，在 auth_user 表中，id=5 的记录就是在调试上述功能时注册的，对应在 account_userprofile 表中，也存在 user_id=5 的记录。

2.4.3　管理新增的注册内容

还记得前面曾经创建了本项目的超级管理员吗？在项目运行起来之后，访问 http://localhost:8000/admin，用超级管理员账户登录，进入可对全站内容进行管理的界面，比如管理网站用户。

在我们增加用户注册信息后，所增加的内容能不能也在超级管理员界面中进行集中管理呢？要实现这个功能，需要对./account/admin.py 文件进行编辑，代码如下。

```
from django.contrib import admin
from .models import UserProfile

class UserProfileAdmin(admin.ModelAdmin):
    list_display = ('user', 'birth', 'phone')
    list_filter = ("phone",)

admin.site.register(UserProfile, UserProfileAdmin)
```

保存代码之后，登录 http://localhost:8000/admin/，会看到如图 2-24 所示的界面。

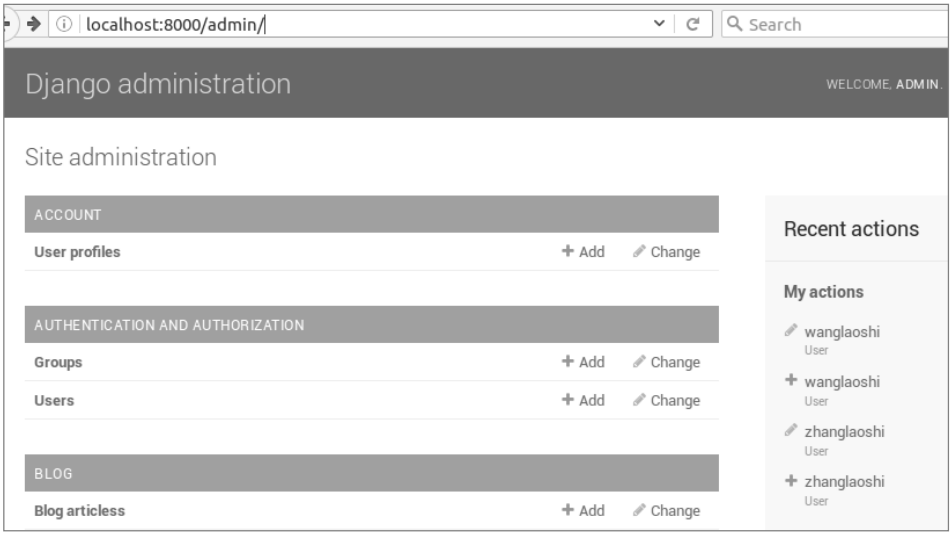

图 2-24　超级管理员界面

其中，ACCOUNT 选项和其下面的 User profiles 选项是由 admin.py 中的代码新增加的，单击 User profiles 选项，可以看到对新增加的注册内容的管理，如图 2-25 所示。

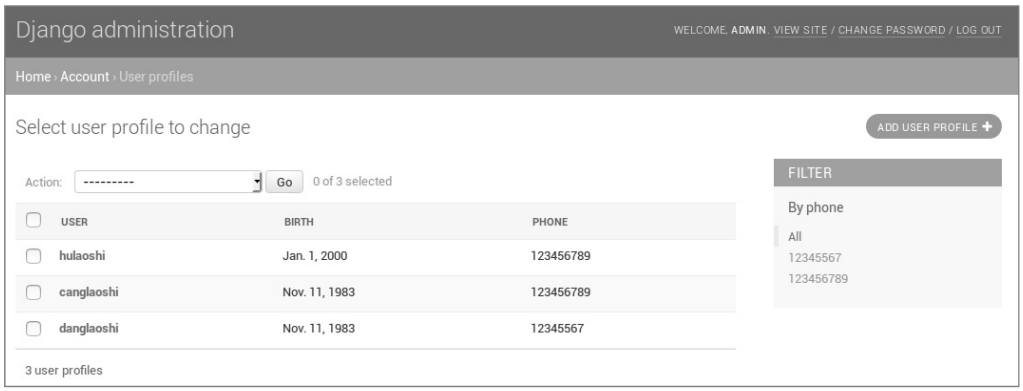

图 2-25　用户管理界面

对照此结果，对刚才所编写的 UserProfileAdmin()函数进行简要说明。随着后续学习，还会不断深化对类似功能的理解。

- list_display 属性的作用是列出列表中的项目。注意这个属性的名称不能改变，后面所有类似的属性也是如此。
- list_filter 的作用是规定网页右边 FILTER 的显示内容，读者可以根据这里的电话号码过滤显示列表的内容。

当然，作为超级管理员也有权限增加用户的信息，单击 ADD USER PROFILE 按钮可以进行增加用户信息的操作。

通过上述操作可以看出，在 admin.py 文件中所编写的代码，主要是将某些内容注册到后端管理界面，从而让超级管理员对该内容进行管理。

注册功能对网站的重要性不言而喻，不同的网站会根据自己的要求对注册功能提出不同的需求。读者在自己的项目中，可以使用类似本项目的方式实现，也可以不使用 Django 默认的 User 表，完全自己来编写用户管理的相关应用，只不过这时要考虑有关的周边问题，不仅仅是把用户的信息保存到数据库，比如 session 等。

2.4.4　知识点

1．模型：字段属性类型

数据模型类（Model）依然是 Python 中的类，可以在其中命名属性（变量）和方法。每个属性（变量）对应着数据库中的一个字段，所以在 Django 的数据模型类中，也将变量称为字段。数据库中的字段要有自己的属性，那么数据模型类中的字段也应该有属性，这些属性在 django.db.models 中，比如 title = models.CharField(max_length=300)。

在默认情况下，Django 会为每个数据模型类设置一个自增的 id 字段，并将其作为该数据库表的主键，因此这个字段不需要在数据模型类中写出来。当然，开发者也可以在数据模型类中自行指定主键，方法是在该字段属性中增加 primary_key=True。除这个默认的 id 字段外，其他

字段都需要开发者在数据模型类中声明其名称和属性。

- CharField：用于保存字符串，在使用时一定要声明字符串的长度。
- TextField：与 CharField 一样，区别在于保存的字符串长度是无限的（严格说应该受制于数据库程序和硬件系统），通常用于保存较大文本。
- EmailField、URLField：都继承了 CharField，只不过其中包含了验证它们的值是否是 E-mail 和 URL 的方法。将某个字段设置为该类型，不需要再编写对写入数据的验证方法。
- FileField：表示该字段接收上传文件，同时将上传的文件保存到服务器中。
- DateField、DateTimeField：用于保存时间，有一个常用参数 auto_now_add，如果 auto_now_add=True，那么当数据模型实例被保存时，当前时间将自动被存储，而不需要为该字段进行赋值。

换个角度看，每个字段其实也是建立了一个属性类的实例，比如 title = models.CharField (max_length=300)，CharField 就是一个类，以这种方式建立了一个实例，并赋值给变量 title。在创建实例时，传入类的参数会有所不同，从而创建不同的实例，比如可以使用 unique=True，即意味着该实例或者该字段在数据库中的值必须是唯一的（但不一定是主键）。比如用户名或者用户邮箱，就可以做如此设置。

关于字段属性，上述仅列出几项，随着项目深入读者还会遇到很多。

2．用户注册拓展知识

Django 默认的 django.contrib.auth 应用中有登录 login()方法，但是没有用户注册的方法，为什么没有？反正没有，这是现实，就别追问原因了，不管怎样，都要自己动手。本节介绍了一种简单的用户注册过程，所采用的数据模型是 Django 内置的用户管理应用中的数据模型（from django.contrib.auth.models import User），也就是将用户注册信息存入上节所使用的数据库表中。

用户注册是一个比较个性化的功能，如果所需要的用户注册信息超出了默认数据模型的范畴，还要自己完成扩展部分，本节也做了这种演示。

虽然 Django 没有默认的注册功能，但是注册是网站开发所不可或缺的。全世界众多的 Django 开发者对这种常用而 Django 官方不提供的功能往往是痛心疾首的，于是奋起改变，就出现了很多第三方的 Django 应用，比如针对注册的 django-registration 就是一个典型例子，读者可以访问官方网站（http://django-registration.readthedocs.io/）了解这个应用的使用方法，可以按照网站的说明自己进行调试。

现在是社交媒体泛滥的时代，很多网站都支持用社交媒体账号登录，用 Django 开发的网站项目当然也能做到。要完成这个工作，开发者可以根据某个社交媒体官方提供的 API，结合自己的网站项目完成。Django 继承了 Python 的优良传统——开发，这就让很多人有机会为 Django 提供“轮子”。所以，如果需求是某种比较通用的功能，可以先在网上找找有没有“轮子”。Django Social Auth 就是一个专门用于社交媒体账户登录 Django 项目的第三方应用，github.com 上的源码地址是 https://github.com/omab/django-social-auth，其中有较为详细的说明。建议读者在基本完成本书的学习之后，再来学习这个内容。

3．文档导读

（1）*Model field reference*，https://docs.djangoproject.com/en/2.1/ref/models/fields/。

（2）用 django-social-auth 做中国社交网站三方登录，http://blog.csdn.net/asd131531/article/details/42457389?locationNum=10&fps=1。

2.5 关于密码的操作

密码与我们形影不离，简直都成为记忆负担了。网站上一般都要有关于密码的操作，常见的就是重置密码、修改密码和找回密码。关于密码的操作如图 2-26 所示。

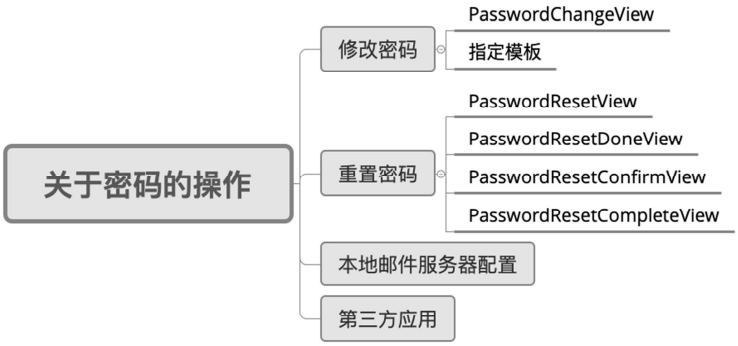

图 2-26　知识技能导图

2.5.1 修改密码

在 2.3 节中，曾经使用 Django 内置的 LoginView 实现了登录功能，LoginView 是一种基于类的视图。与此类似，Django 也提供了修改密码的默认方法，并且在 Django 2.x 中都是基于类的了。

在进行具体操作之前，先梳理清楚修改密码的步骤：

第一步，要判断用户是否登录，只有登录用户才能修改密码。

第二步，要打开设置新密码的界面，在该界面中提交新密码。

第三步，要给用户反馈，是否修改完成。

下面就依据上述三个步骤，利用 Django 内置的方法实现修改密码的功能——既然是使用内置的，显然不需要在视图文件（view.py）中写程序了。

先写上述第二步中用于提交新密码的界面，即在./templates/account/中创建 password_change_form.html 文件。以下是./templates/account/password_change_form.html 的代码。

```
{% extends "base.html" %}
{% block title %}passowrd change{% endblock %}
{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Change Password</h1>
    <p>Please enter your old password, for security's sake, and then enter your new password
```

```
twice so we can verify you typed it in correctly.</p>
    {% if form.new_password1.help_text %}
    <div class="text-left" style="margin-left:400px">
    <p>{{ form.new_password1.help_text|safe }}</p>
    </div>
    {% endif %}
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="form-group">
            <label class='col-md-5 control-label text-right'>
            {{ form.old_password.label_tag }}
            </label>
            <div class="col-md-6 text-left">{{ form.old_password }}</div>
        </div>

        <div class="form-group">
            <label class='col-md-5 control-label text-right'>
            {{ form.new_password1.label_tag }}
            </label>
            <div class="col-md-6 text-left">{{ form.new_password1 }}</div>
        </div>

        <div class="form-group">
            <label class="col-md-5 control-label text-right">
            {{ form.new_password2.label_tag }}
            </label>
            <div class="col-md-6 text-left">{{ form.new_password2 }}</div>
        </div>
        <input type="submit" value="Change my password" class="btn btn-primary btn-lg"
/>
    </form>
</div>
{% endblock %}
```

同样，也写好第三步中密码修改完成的页面，即./templates/account/password_change_done.html，代码如下。

```
{% extends "base.html" %}
{% block title %}password change done{% endblock %}
{% block content %}
<div class="row text-center vertical-middle-sm">
<p>Your password was changed.</p>
</div>
{% endblock %}
```

接下来要做的就是配置好 URL。在./account/urls.py 中增加如下 URL 配置项。

```
urlpatterns = [
    …
path('password-change/',
         auth_views.PasswordChangeView.as_view(
                template_name="account/password_change_form.html",
                success_url="/account/password-change-done/"),
         name='password_change'),
```

```
path('password-change-done/',
        auth_views.PasswordChangeDoneView.as_view(
               template_name="account/password_change_done.html"),
        name='password_change_done'),
]
```

以上工作完成之后，重新启动 Django 服务。先打开 http://localhost:8000/blog/页面，确保有一个用户已经登录网站，然后另外打开一个浏览器标签，输入 http://localhost:8000/account/password-change/，试图进入修改密码的界面。

再次强调，此时如果要出现图 2-27 所示的修改密码界面，必须是先实现了用户登录。

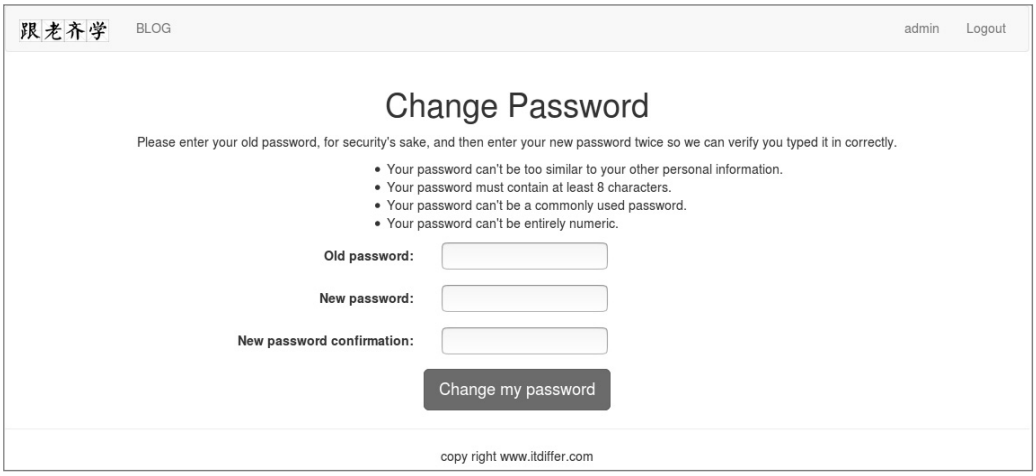

图 2-27　修改密码页面

输入当前用户的密码以及修改之后的新密码，单击 Change my password 按钮，跳转到如图 2-28 所示的页面，说明密码修改成功。

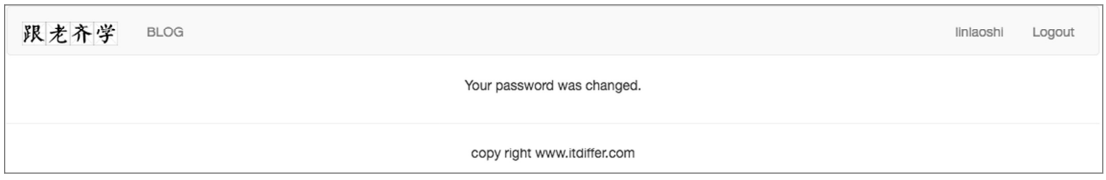

图 2-28　密码修改成功

如果成功了，请再耐心地看一看在配置/account/password-change 的 URL 时所调用的方法 auth_views.PasswordChangeView.as_view()，我们向它提供了两个参数：template_name 说明了修改密码的页面模板；success_url 则说明了如果密码修改成功，页面跳转对象，即/account/password-change-done/。同理，还要用类似的方法配置这个地址。

前面强调过，要完成密码的修改，用户必须登录。如果用户没有登录，访问了 http://127.0.0.1:8000/account/password-change/会怎样呢？要大胆尝试。

图 2-29 所示为没有登录的时候访问修改密码的地址出现的 404 错误。当用户发起修改密码的请求之后，Django 首先要判断用户是否登录，如果没有登录，就拒绝返回页面。但是，要注

意观察图 2-29 中显示的 Request URL，这是一个重要信息，这里显示的不完全等同于我们所访问的地址：http://localhost:8000/accounts/login/?next=/account/password-change/，此前并没有在 URL 中配置这个地址（注意/accounts/不是本项目中的/account/），并且在地址中还有一个 next，为何如此？

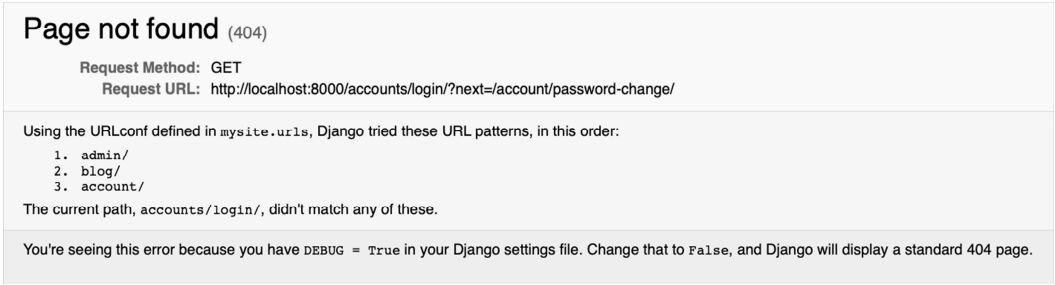
Page not found (404)
Request Method: GET
Request URL: http://localhost:8000/accounts/login/?next=/account/password-change/
Using the URLconf defined in mysite.urls, Django tried these URL patterns, in this order:
1. admin/
2. blog/
3. account/
The current path, accounts/login/, didn't match any of these.
You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.

图 2-29　用户未登录访问修改密码页面

Django 已经判断出用户没有登录，用上面的方式表示要首先访问登录界面，即/accounts/login/，当用户登录之后，再自动跳转到 next 所指示的目标，即/account/password-change/。这是一个非常好的设计，并且在很多网站中能看到。比如到某网店中，如果选中了某个商品，点击“立即购买”，这时网站系统就会先判断你是否登录，如果没有登录，会跳出登录窗口令你登录，待登录之后自动回到你所选定的商品页面。

只不过，上面报错是因为/accounts/login/不是本项目中设定的登录 URL，它是 Django 默认的，本项目中的登录 URL 是/account/login/——我故意这么做的目的就是引出后面的操作。

如何让 Django 能够跳转到/account/login/呢？请编辑./mysite/settings.py 文件，在文件中增加如下内容。

```
LOGIN_URL = '/account/login/'
```

然后，再次在没有登录的情况下访问 http://127.0.0.1:8000/account/password-change/，是否自动跳转到登录界面了？

如果上面设置无误，肯定已经进入到登录界面了。填写用户名和密码之后，完成登录。但是，并没有跳转到修改密码界面。

为了实现再次跳转，还需要编辑./templates/account/login2.html 文件（即登录模板），该文件新增代码如下。

```
{% block content %}
<div class="row text-center vertical-middle-sm">
    … #省略部分代码

    <p>Input your username and password</p>

    {% if next %}    <!—新增-->
    <form class="form-horizontal" action="{% url 'account:user_login' %}?next={{next}}"
method="post" >   <!—新增-->
    {% else %}    <!—新增-->
    <form class="form-horizontal" action="{% url 'account:user_login' %}" method="post">
```

```
    {% endif %}    <!—新增-->
        {% csrf_token %}
        <div class="form-group">
            …  #省略部分代码
        </div>
        <input type="submit" class="btn btn-primary btn-lg" value="Login">
    </form>
</div>
{% endblock %}
```

按照上面的方式修改了登录模板文件之后，再次测试，就能够实现登录之后跳转到修改密码页面的功能了。

请读者对上述流程反复揣摩，因为这种“访问页面—跳转登录—跳转页面”的实现不仅仅用于修改密码，更多地可以用于其他功能之中。

对于一个网站来说，总要有一个地方能够让用户进入修改密码的页面，这就是入口。在用户登录之后，导航条上会显示该用户的用户名，可以将修改密码的入口放在这里。单击用户名，出现一个下拉菜单，单击“修改密码”，跳转到刚才已经建立的修改密码页面。

所要修改的仅仅是一个模板文件./templates/header.html，修改之后其代码如下。

```
{% load staticfiles %}
#省略其余代码
{% if user.is_authenticated %}
<li><!—以下为新增-->
<div class="dropdown" style="margin-top:8px">
       <button class='btn btn-default dropdown-toggle' type='button' id='dropdownMenu'
data-toggle='dropdown'>{{ user.username }}
           <span class='caret'></span>
       </button>
       <ul class="dropdown-menu">
           <li><a href="{% url 'account:password_change' %}">修改密码</a></li>
       </ul>
   </div><!—修改结束-->
</li>
#省略部分代码，并在本文件的末端增加对 JavaScript 文件的引入
<script src="{% static 'js/jquery.js' %}"></script>
<script src="{% static 'js/bootstrap.js' %}"></script>
```

上述代码中<script>部分不要丢失，因为我们在修改密码的功能上使用了 bootstrap 提供的下拉菜单功能，需要 JavaScript 脚本支持。

修改好密码后，检查 Django 服务是否启动，并且保证用户处于退出状态。然后重新登录，在导航条的右侧显示用户名的位置单击鼠标，在下拉菜单中就能看到“修改密码”了，如图 2-30 所示。

图 2-30　修改密码入口

单击“修改密码”，就会跳转到前面已经做好的修改密码页面，用户可以在该页面中修改自己的密码。

2.5.2　重置密码

如果用户忘记了密码怎么办？找回是不可能的，因为网站对用户的密码都进行了加密，而且好的加密方法都是不可逆的，常用的加密算法有 MD5、SHA1 等（当然，破解加密方式也是一个研究领域）。

Django 默认的用户管理系统也自带了加密模块。比如本书前述章节的实例中，用户注册、登录时要输入密码，读者注意观察数据库中用户的密码，显示的不是在键盘上所输入的字符，而是加密后的效果。

如果这个密码被用户忘记了，就不能“找回”，只能“重置”。

Django 提供了内置的方法，实现“重置”密码的功能。流程与登录、修改密码的操作类似，只是过程稍微烦琐。所以，读者要慢慢阅读。

图 2-31 所示为重置密码的流程，下面就按照这个流程完成此功能的开发。

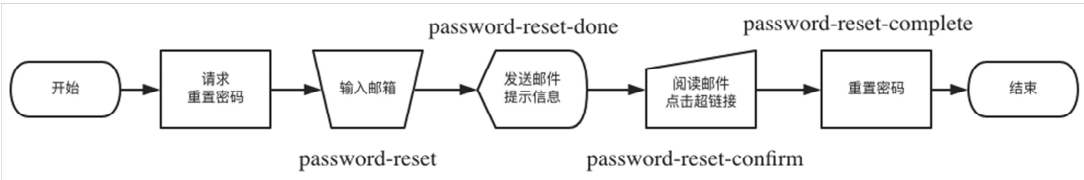

图 2-31　重置密码的流程

对用户请求重置密码的响应，使用 Django 内置的 PasswordResetView 类视图完成，即在./account/urls.py 中配置如下 URL 映射。

```
path('password-reset/',
     auth_views.PasswordResetView.as_view(
               template_name="account/password_reset_form.html",
               email_template_name="account/password_reset_email.html",
               success_url='/account/password-reset-done/'),
     name='password_reset'),
```

PasswordResetView.as_view 方法的 3 个参数的含义如下：

- template_name="account/password_reset_form.html"：设置请求重置密码的前端页面，需要在./templates/account/中新建 password_reset_form.html 文件，其代码如下。

```
{% extends "base.html" %}
{% block title %}passowrd change{% endblock %} {% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Change Password</h1>
    <p>Please enter your old password, for security's sake, and then enter your new password
twice so we can verify you typed it in correctly.</p>
    {% if form.new_password1.help_text %}
    <div class="text-left" style="margin-left:400px">
        <p>{{ form.new_password1.help_text|safe }}</p>
```

```
    </div>
    {% endif %}

    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="form-group">
            <label class='col-md-5 control-label text-right'>
                {{ form.old_password.label_tag }}
            </label>
            <div class="col-md-6 text-left">{{ form.old_password }}</div>
        </div>

        <div class="form-group">
            <label class='col-md-5 control-label text-right'>
                {{ form.new_password1.label_tag }}
            </label>
            <div class="col-md-6 text-left">{{ form.new_password1 }}</div>
        </div>

        <div class="form-group">
            <label class="col-md-5 control-label text-right">
                {{ form.new_password2.label_tag }}
            </label>
            <div class="col-md-6 text-left">{{ form.new_password2 }}</div>
        </div>

        <input type="submit" value="Change my password" class="btn btn-primary btn-lg"/>
    </form>
</div>
{% endblock %}
```

- email_template_name="account/password_reset_email.html"：设置向申请重置密码的用户邮箱所发送的邮件内容，同样要创建新的文件./templates/account/password_reset_email.html，代码如下。

```
<p>You're receiving this email because you requested a password reset for your user account
at <a href="http://www.itdiffer.com">itdiffer.com</a></p>
<p>Please go to the following page and choose a new password:</p>
{{ protocol }}://{{ domain }}{% url 'account:password_reset_confirm' uidb64=uid
token=token %}
<p>Your username, in case you've forgotten:{{ user.get_username }}</p> <p>Thanks for using
our site!</p>
<p>The itdiffer.com team</p>
```

- success_url='/account/password-reset-done/'：向用户发送邮件后，当前页面跳转目标。所以，还要继续在./account/urls.py 中增加 URL 配置项。

```
path('password-reset-done/',
     auth_views.PasswordResetDoneView.as_view(
                    template_name="account/password_reset_done.html"),
     name='password_reset_done'),
```

在此处又出现了 password_reset_done.html，还需要在./templates/account/中创建该文件，并录入以下代码。

```
{% extends "base.html" %}
{% block title %}password reset{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Reset your password</h1>
    <p>We've emailed you instructions for setting your password, if an account exists with
the email you entered. You should receive them shortly.</p>
    <p>If you don't receive an email, please make sure you've entered the address you
registered with, and check your spam folder.</p>
</div>
{% endblock %}
```

上述配置完成，即实现了用户提交个人邮箱，并向该邮箱发送重置密码的邮件功能。请读者观察一下所发送的邮件内容，即 password_reset_email.html，会发现其中有“{{ protocol }}://{{ domain }}{% url 'account:password_reset_confirm' uidb64=uid token=token %}”，这就是让用户点击并打开新页面，在此页面输入新密码的超级链接。这个链接怎么生成的呢？还是要在./account/urls.py 中增加如下 URL 配置。

```
path('password-reset-confirm/<uidb64>/<token>/',
     auth_views.PasswordResetConfirmView.as_view(
                  template_name="account/password_reset_confirm.html",
                  success_url='/account/password-reset-complete/'),
     name="password_reset_confirm"),
```

依然是为 PasswordResetConfirmView.as_view()提供两个已经很熟悉的参数，template_name 的值是一个模板文件，在./templates/account/中创建 password_reset_confirm.html 文件，代码如下。

```
{% extends "base.html" %}
{% block title %}password reset{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">

    <h1>Reset Password</h1>
    <p>Please enter your new password twice so we can verify you typed it in correctly.</p>
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="form-group">
            <label class='col-md-5 control-label text-right'>New password</label>
            <div class="col-md-6 text-left">{{ form.new_password1 }}</div>
        </div>
        <div class="form-group">
            <label class='col-md-5 control-label text-right'>Confirm password</label>
            <div class="col-md-6 text-left">{{ form.new_password2 }}</div>
        </div>
        <input type="submit" value="Change my password" class="btn btn-primary btn-lg"
/>
    </form>
</div>
{% endblock %}
```

当用户点击收到的邮件中的超链接时，就会打开以上述文件为前端的网页，在该页面中输入新密码。新密码输入完毕后，就会跳转到参数 success_url 的值所指示的目标对象，它是另外一个 URL 配置，所以要继续编辑./account/urls.py，增加如下内容。

```
path('password-reset-complete/',
     auth_views.PasswordResetCompleteView.as_view(
               template_name='account/password_reset_complete.html'),
     name='password_reset_complete'),
```

这里的 PasswordResetCompleteView.as_view()中的参数 template_name 也是以模板文件为值。于是，要继续在./templates/account/中创建文件 password_reset_complete.html（代码如下），当用户最终完成新密码提交之后，就跳转到以这个文件为前端显示的页面。

```
{% extends "base.html" %}
{% block title %}password reset{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Reset your password</h1>
    <p>Your password has been set. You may go ahead and <a href="{% url
'account:user_login' %}">log in now</a>.
    </p>
</div>
{% endblock %}
```

进展还是比较顺利的，这时可以重新启动 Django 服务，打开浏览器，输入 http://localhost:8000/account/password-reset/，效果如图 2-32 所示。

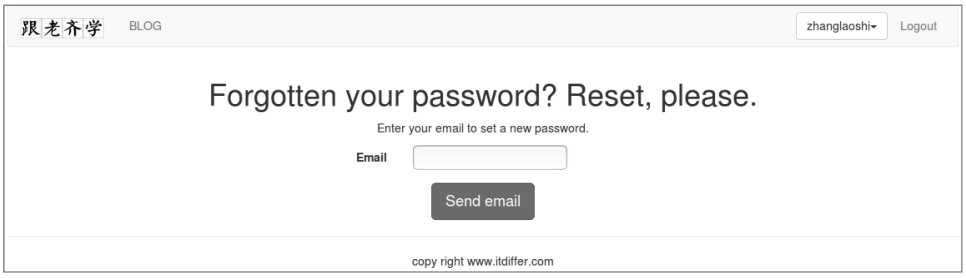

图 2-32　重置密码页

在这个页面中输入自己的邮箱（注意，从现在开始，在调试重置密码的时候，要使用目前系统中已注册用户的有效邮箱，读者可以通过管理员增加有关用户及其有效邮箱），然后单击 Send email 按钮。

你的界面显示效果是什么？是报错吗？还是其他提示信息？

笔者的计算机中出现了如图2-33所示的报错页面，如果你的页面不是这样的，也不要高兴，到刚才输入的邮箱里面看一下。

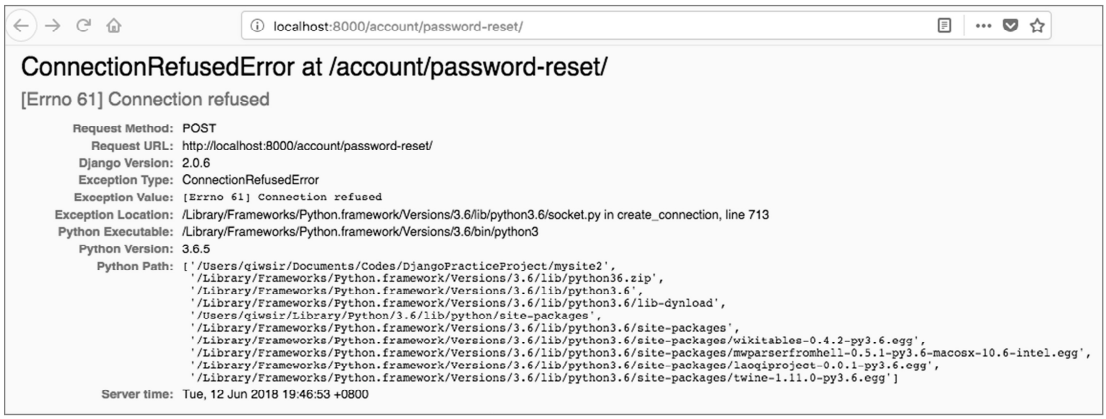

图 2-33　邮件发送报错

看来没有那么简单，开发一定要脚踏实地，不能浮光掠影地理解个大概。有人说“细节决定成败”，虽然决定成败的因素很多，“细节”仅是诸多因素之一，但在编程中，细节是必须要认真考虑的。下面就要认真研究到底是什么原因导致这个错误。

我们的意图是在一个服务器（虽然是开发用的计算机，但 Django 服务已经运行，也就被认为是服务器了）上向另外一个服务器发送邮件。要实现邮件发送功能，必须要配置邮件发送服务器。通常，不需要自己搭建邮件服务器，可以使用目前已有的，比如此处示例使用网易邮箱发送邮件，需要有一个真实的网易邮箱。

然后，在./mysite/settings.py 文件中进行如下配置。

```
EMAIL_HOST = 'smtp.163.com'
EMAIL_HOST_USER = "your_account@163.com"
EMAIL_HOST_PASSWORD = "your_password_email"
EMAIL_PORT = 25
EMAIL_USE_TLS = True
DEFAULT_FROM_EMAIL = "your_account@163.com"
```

请读者注意，用于上述调试的邮箱，不要设置“客户端授权密码”。另外，当读者阅读本书时，或许上述有关邮箱提供服务商已经修改了针对客户端的邮箱服务规则，请读者根据其官方说明对上述配置进行修改。

上述配置完成之后，再次访问 http://localhost:8000/account/password-reset/页面，输入本系统中一个真实账号的邮箱，然后单击 Send email 按钮（见图 2-32），界面就跳转到如图 2-34 所示的效果。

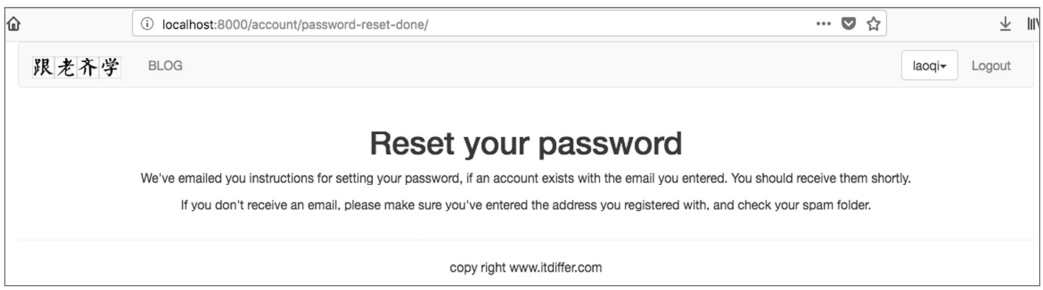

图 2-34　重置密码邮件发送成功

这表明向某用户的邮箱发送了一封邮件。登录邮箱看一下，是不是有刚才这封邮件呢？

邮件收到，如图 2-35 所示，表明配置成功。

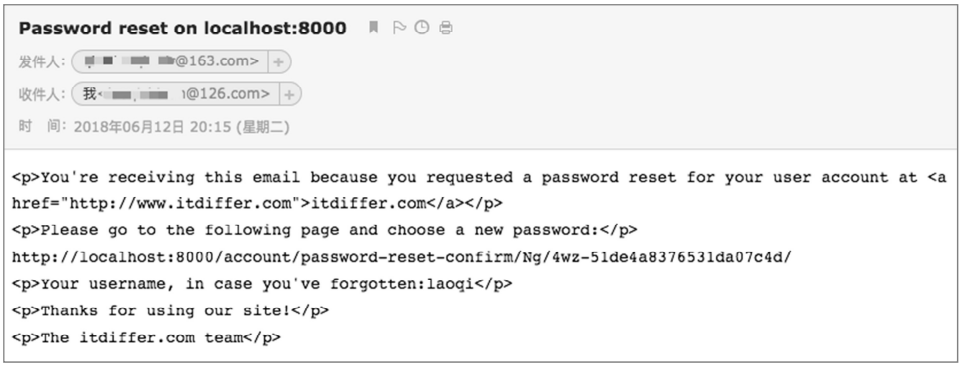

图 2-35　收到重置密码的邮件

注意，因为各种原因，包括开发环境和各项配置，也包括所使用邮箱的某些配置和该网站的规定，笔者不保证读者按照上面的顺序操作就能够顺利成功。如果遇到了错误，可以通过 Google 解决。请坚信，你的错误别人都犯过，而且被解决了。

回到浏览器中。

将图 2-35 中提示重置密码所用的链接 http://localhost:8000/account/password-reset-confirm/Ng/4wz-51de4a8376531da07c4d/复制出来，打开该页面，如图 2-36 所示。

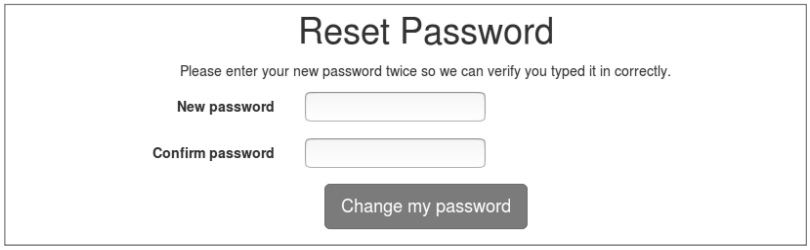

图 2-36　输入新密码

输入新的密码，注意在 Django 中，密码不能太简单。新密码输入完毕，单击 Change my password 按钮，显示重置成功的结果页面，如图 2-37 所示。

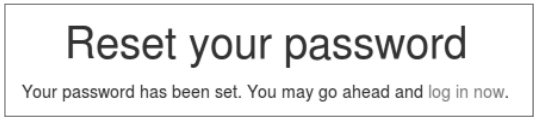

图 2-37　密码重置成功

在开发过程中，常常习惯随便输入一个邮箱地址，比如前面笔者曾经使用了 zhang@laoshi.com 邮箱，但是，事实上没有地方去收这个邮件。为了便于开发，可以在./mysite/settings.py 中增加一行代码，追加到末尾即可，代码如下。

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

这行代码的作用就是将发送的邮件直接显示在控制台，如图 2-38 所示，而不是发送到指定邮箱，这种做法在开发调试过程中非常有用。但是将程序部署到生产环境中时，一定要记着，这行代码要删掉，否则用户收不到邮件。

图 2-38　在控制台上显示邮件发送效果

以上重置密码过程使用了 Django 的内置方法。

在 Django 领域，不仅有官方提供的内置方法可供开发者使用，还有大量的第三方应用提供给开发者选择，这就让从事 Django 开发的程序员有了更多的“轮子”可选，所以用 Django 做网站，“轮子”多、选择广、开发效率高，比如重置密码的应用 django-password-reset。但是在本书修订时候它已经支持 Django 2.x，读者可以关注其官方网站（https://django-password-reset.readthedocs.io/en/latest/），建议尝试使用。

2.5.3　知识点

1．密码强度

维基百科中如此定义“密码强度”：指一个密码对抗猜测或暴力破解的有效程度。一般来说，指一个未授权的访问者得到正确密码的平均尝试次数。密码的强度和其长度、复杂度及不可预测度有关。强密码可以降低安全漏洞的整体风险，但并不能降低采取其他安全措施的需要。

通常在网站上，许可用户使用多种字符来设置密码，从而尽可能提高密码强度。有的用户设置的密码比较简单，必要时网站应该对这种简单的设置进行初步判断，比如在页面端检测用户设置密码的长度等。

下面是一些常见的简单密码（源自维基百科的总结），不知道读者是否躺着中枪了。

- 顺序或重复的字符：“12345678”“111111”“abcdefg”“asdf”“qwer”（键盘上的相邻字母）。
- 使用数字或符号的仅外观类似替换，例如使用数字“1”“0”替换英文字母“i”“O”，字符“@”替换字母“a”等。
- 登录名的一部分：密码为登录名的一部分或完全与登录名相同。
- 常用的单词或者汉语拼音：如自己和熟人的名字及其缩写、常用的单词及其缩写、宠物的名字等。
- 常用数字：比如自己或熟人的生日、证件编号等，以及这些数字与名字、称号等字母的简单组合。

2．Django 的密码管理

在正常情况下，网站应该使用不可逆加密算法来为用户提交的密码加密，比较常用的这类算法有（不仅仅限于如下几种）：

- MD5；
- PBKDF2；
- Bcrypt。

如果查看本项目已经创建的数据库表 auth_user（Django 默认的用户管理应用的数据库表），会发现每个注册用户的密码都是一大串字符串，虽然我们在完成用户注册功能时，并没有直接使用有关“加密”的函数，但 Django 依然为我们默默地将用户的密码加密了，如图 2-39 所示（利用 SQLite 可视化工具查看的结果截图）。

TABLE auth_user　Search　Show All　Add　Duplicate　Edit　Delete

id	password	last_login	is_super…	first_name	last_name	email	is_
1	pbkdf2_sha256$30000$CxK72Fd72Dp6$XsHr/R5P3h9MaHVSSCiwwJ…	2017-04-01 0…	1			admin@adm…	1
2	pbkdf2_sha256$30000$WvQRizRNgL1v$ysWnCx5Hle7plCPcy/ITQXN…	2016-11-25 1…	0	zhang	laoshi	zhang@laos…	0
3	pbkdf2_sha256$30000$L3jN5EA5oeFh$LabxFJXndQl8UjHtW5fwh/IIx…		0	wang	laoshi	wang@laosh…	0
4	pbkdf2_sha256$30000$SnjRB0kL764y$C9OETO3WzRgzgo5lZILMFten…		0				0
5	pbkdf2_sha256$30000$wNbFtnrgdhVt$lCNxHZp7os6uC0uOQ3u4uH…		0				0
6	pbkdf2_sha256$30000$H0OYDk8pjMIH$0j8zeYT8nXx7Z3celTN+hwF…		0				0
7	pbkdf2_sha256$30000$4qmUvawBfAGh$Ko4J7Hhs4+2ILXzeYzK/b8e…		0				0
8	pbkdf2_sha256$30000$vceLxuxl4opZ$f2t8wPlsFliYlVJK92UYhjiG1ly…	2016-11-13 1…	0			laoqishixun…	0
9	pbkdf2_sha256$30000$k7q3j7UFDN0i$V4wfSpCFOJDaiL1RUa3PDtZ…	2016-11-25 1…	0			hu@laoshi.c…	0
10	pbkdf2_sha256$30000$1fRDjOKohcu$UkX7XKMHuxKoKU+bVLno19…	2017-03-16 0…	0			yin@laoshi.c…	0
11	pbkdf2_sha256$30000$l2t73MYwToPr$5LUSHJOMKL/+/zwSZGQf1Ir…	2017-03-30 0…	0			lu@laoshi.com	0
12	pbkdf2_sha256$30000$1RyhfHHQOBFn$BlHa/O03jzkXArtMVqHaPm…		0			chen@laoshi…	0
13	pbkdf2_sha256$30000$aQ0qxUJ1xQYP$qIRWxjXp61OkbhzSyJMVeO…		0			kuang@laos…	0

图 2-39　用户和密码

Django 默认使用 PBKDF2（Password-Based Key Derivation Function2）加密算法，有人专门撰文研究了破解这类密码所耗费的时间（原文网址：https://blog.agilebits.com/2014/03/10/crackers-report-great-news-for-1password-4/），如图 2-40 所示。

Password Strength			Average time to guess at (pessimistic) 20,000 guesses per second	Average time to guess at 5,000 guesses per second
Bits	Letters†	Words‡		
39	7	3	136 days	544 days
52	9	4	2,890 years	11,562 years
65	11	5	22,476,384 years	89,905,535 years
78	14	6	175 billion years	699 billion years
90	16	7	1,359 trillion years	5,436 trillion years

“Billion” = 10^9; “trillion” = 10^{12}

†Length in mixed cased letters (52 possibilities) chosen *completely at random*

‡Length in words from the Diceware list chosen *completely at random*

图 2-40　破解密码比较

在实际的项目开发中，或许因为某种原因，要使用其他的加密算法，对此 Django 官方文档有专门说明。

3．文档导读

（1）*Password management in Django*，https://docs.djangoproject.com/en/dev/topics/auth/passwords/。

（2）*A Django application that provides a login form with a Remember Me checkbox*，https://github.com/jimfmunro/django-remember_me/。

2.6　维护个人信息

在用户完成网站注册和登录之后，大多数网站要求用户提供更多的个人信息，因此开发在网站上维护用户个人信息的功能是很有必要的。同时，读者也可以在该功能编写过程中体验另外一种前后端传递数据的方式，如图 2-41 所示。

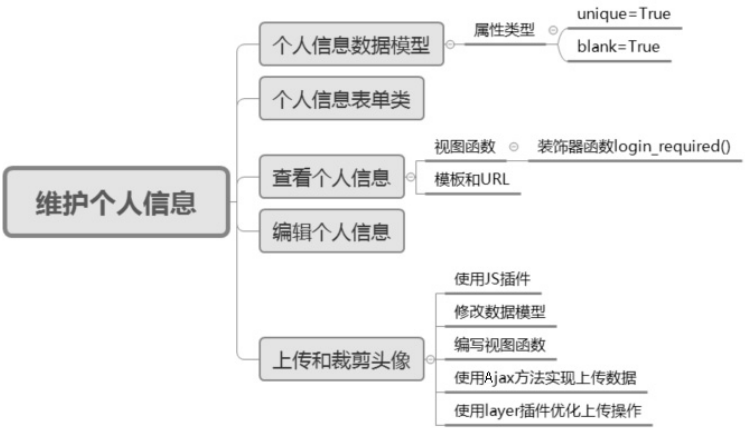

图 2-41　知识技能导图

2.6.1　个人信息的数据模型类和表单类

所有的个人信息都要保存到数据库中，因此要编辑./account/models.py 文件，创建个人信息相关的数据模型类，在这个文件中增加下面的代码。

```
class UserInfo(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE, unique=True)
    school = models.CharField(max_length=100, blank=True)
    company = models.CharField(max_length=100, blank=True)
    profession = models.CharField(max_length=100, blank=True)
    address = models.CharField(max_length=100, blank=True)
    aboutme = models.TextField(blank=True)    #①

    def __str__(self):
        return "user:{}".format(self.user.username)
```

上述数据模型类与以前的数据模型类相比区别不大，需要说明的是语句①中的 TextField 属性，它将对应 HTML 中的 Textarea。在多个字段中都有 blank=True，其含义是该项数据可以为空，即在前端界面中允许用户不填写。

数据模型类创建完成，下面迁移数据，让数据库具有相应的数据库表。

```
$ python3 manage.py makemigrations account
Migrations for 'account':
  account/migrations/0002_userinfo.py
    - Create model UserInfo

$ python3 manage.py migrate
Operations to perform:
```

```
  Apply all migrations: account, admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying account.0002_userinfo... OK
```

数据模型类创建之后，后面的开发步骤类似于 2.4 节所学习过的“注册”过程，依次是“编写表单类”“创建视图函数”“设置 URL”“设置前端模板”，下面就按部就班地逐一展开。

表单类还是要写在./account/forms.py 文件中，下面是在此文件中增加的代码。

```
from .models import UserProfile, UserInfo
#其他代码从略
class UserInfoForm(forms.ModelForm):
    class Meta:
        model = UserInfo
        fields = ("school", "company", "profession", "address", "aboutme")
```

上面的表单类是针对 UserInfo 数据模型类的，对于来自 auth_user 数据库表的信息，也要写一个表单类来对应，因为有必要对这个数据库表的数据进行修改，这就是下面的 UserForm 类要完成的工作了。

```
class UserForm(forms.ModelForm):
    class Meta:
        model = User
        fields = ("email",)
```

特别注意，在 UserForm 的 fields 值中不包括 username，因为用户的 username 一旦确定就不能随便更改。在用户编辑个人信息时，username 的值不再是可修改的，或者说在本项目中约定用 username 为用户的唯一标识，通常唯一标识就不让用户修改了。如果允许用户修改，就要换别的字段为唯一标识，比如用户 ID。

至此，我们所做的都是熟悉的流程，还没有牵扯到新的知识，复习旧的知识也非常重要，正所谓温故知新。读者不妨现在闭上眼睛，回忆前面的开发流程，检验自己是不是熟悉了 Django 的基本开发过程，一定要闭上眼睛想！

2.6.2 展示个人信息

按照习惯的流程，接下来就是编写视图函数了。这里要编写的视图函数应该能够处理两种请求，一种是展示用户信息的 GET 请求，另一种是前端提交用户信息的 POST 请求。

要展示用户的个人信息，必须看一下目前的数据库中都有哪些与个人信息相关的内容。

- account_userinfo 表，其中记录了 school、company、profession、address、aboutme 字段内容。
- account_userprofile 表，其中记录了 phone、birth 字段内容。
- auth_user 表，这是 Django 默认的，其中记录了 password、last_login、si_supperuser、first_name、last_name、email、username 等字段内容。

此外，关于用户信息还有 Django 默认的其他表，如 auth_user_group 等，此处我们暂时用不到。在下面的例子中，只从上述三个数据库表中获取某些字段的内容，用来组成用户的个人信息。

在./account/views.py 文件中增加如下代码，这是与个人信息相关的视图函数。

```
#原来引入的部分内容从略
from django.contrib.auth.decorators import login_required    #①
from .models import UserProfile, UserInfo
from django.contrib.auth.models import User
#其他视图函数从略，下面是展示个人信息的视图函数
@login_required()     #②
def myself(request):
    userprofile = UserProfile.objects.get(user=request.user) if
                          hasattr(request.user, 'userprofile') else
                          UserProfile.objects.create(user=request.user)
    userinfo = UserInfo.objects.get(user=request.user) if
                    hasattr(request.user, 'userinfo') else
                    UserInfo.objects.create(user=request.user)
    return render(request, "account/myself.html",
                {"user":request.user,
                 "userinfo":userinfo,
                 "userprofile":userprofile})
```

因为只有登录的用户才能看自己的个人信息，所以要在该视图函数被执行时判断用户是否登录，此处使用了 Django 自带的装饰器函数，即在前面使用语句①引入的装饰器函数 login_required。在具体使用时，也可以向装饰器函数提供一个参数，如 login_url='/account/login/'将没有登录的用户转到登录界面。但是，这样的功能本项目中已经在 settings.py 中配置过了（详见 2.5 节），所以此处就不需要给 login_required()提供参数了。

与此视图函数对应的模板应该放在./templates/account 目录中，命名为 myself.html，其代码如下。

```
{% extends "base.html" %}
{% block title %}my information{% endblock %}
{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>My Information</h1>
    <div class="row">
        <div class="col-md-6" >
            <div class="row">
                <div class="col-md-4 text-right"><span>用户名:</span></div>
                <div class="col-md-8 text-left">{{user.username}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>邮箱:</span></div>
                <div class="col-md-8 text-left">{{user.email}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>生日:</span></div>
                <div class="col-md-8 text-left">{{userprofile.birth}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>电话:</span></div>
```

```
                <div class="col-md-8 text-left">{{userprofile.phone}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>毕业学校:</span></div>
                <div class="col-md-8 text-left">{{userinfo.school}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>工作单位:</span></div>
                <div class="col-md-8 text-left">{{userinfo.company}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>职业:</span></div>
                <div class="col-md-8 text-left">{{userinfo.profession}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>地址:</span></div>
                <div class="col-md-8 text-left">{{userinfo.address}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>个人介绍:</span></div>
                <div class="col-md-8 text-left">{{userinfo.aboutme}}</div>
            </div>
        </div>
        <div class="col-md-6">
            <p>picture</p>
        </div>
    </div>
</div>
{% endblock %}
```

下面还差一步，就是完成 URL 的配置，编辑./account/urls.py 文件，增加相应的 URL。

```
urlpatterns = [
    #... 其他部分从略
    path('my-information/', views.myself, name="my_information"),
]
```

重新启动 Django 服务，然后在浏览器的地址栏中输入 http://localhost:8000/account/my-information/，如果不处于登录状态，会自动跳转到登录界面，这就是在视图函数中判断用户是否登录的装饰器函数起作用了。

能不能如愿以偿地看到个人信息界面呢？

没那么简单。不经历风雨怎能见彩虹，不见 Bug 怎能实现功能？你或许会遇到各种错误，要耐心、细心地检查，排除万难，一定能看到如图 2-42 所示效果。

图 2-42　个人信息页面

除展示外，网站还应该允许用户对个人信息进行编辑。

2.6.3　编辑个人信息

在展示出来的个人信息中，还有不少选项没有填写的呢，应允许用户继续填写。此外，对已经填写的部分，也应允许修改，但前面已经说过了，本项目中的用户名不能修改。

下面还是先编写视图函数，在./account/views.py 中创建一个新的视图函数 myself_edit()，其代码如下。

```
#与本视图函数相关的新增代码
from django.http import HttpResponseRedirect    #①
from .forms import UserProfileForm, UserInfoForm, UserForm

@login_required(login_url='/account/login/')
def myself_edit(request):
    userprofile = UserProfile.objects.get(user=request.user) if
                          hasattr(request.user, 'userprofile') else
                          UserProfile.objects.create(user=request.user)
    userinfo = UserInfo.objects.get(user=request.user) if
                      hasattr(request.user, 'userinfo') else
                      UserInfo.objects.create(user=request.user)

    if request.method == "POST":
        user_form = UserForm(request.POST)
        userprofile_form = UserProfileForm(request.POST)
        userinfo_form = UserInfoForm(request.POST)
        if user_form.is_valid() * userprofile_form.is_valid()
                               * userinfo_form.is_valid():
            user_cd = user_form.cleaned_data
            userprofile_cd = userprofile_form.cleaned_data
            userinfo_cd = userinfo_form.cleaned_data
            request.user.email = user_cd['email']
            userprofile.birth = userprofile_cd['birth']
            userprofile.phone = userprofile_cd['phone']
            userinfo.school = userinfo_cd['school']
            userinfo.company = userinfo_cd['company']
```

```
            userinfo.profession = userinfo_cd['profession']
            userinfo.address = userinfo_cd['address']
            userinfo.aboutme = userinfo_cd['aboutme']
            request.user.save()
            userprofile.save()
            userinfo.save()
        return HttpResponseRedirect('/account/my-information/')    #②
    else:
        user_form = UserForm(instance=request.user)
        userprofile_form = UserProfileForm(initial={"birth":userprofile.birth,
                                                    "phone":userprofile.phone})
        userinfo_form = UserInfoForm(initial={"school":userinfo.school,
                                              "company":userinfo.company,
                                              "profession":userinfo.profession,
                                              "address":userinfo.address,
                                              "aboutme":userinfo.aboutme})
        return render(request, "account/myself_edit.html",
                      {"user_form":user_form,
                       "userprofile_form":userprofile_form,
                       "userinfo_form":userinfo_form})
```

语句①中引入了 HttpResponseRedirect，其作用是实现 URL 的转向，当前的 URL 是 http://localhost:8000/account/edit-my-information/（这个路径在 urls.py 中配置），当用户提交了个人信息并被后端程序验证通过和保存后，就执行语句②，页面转到 http://localhost:8000/account/my-information/，用户能够看到自己的修改结果。

先把已有的用户信息读出来，然后判断用户请求是 POST 还是 GET。如果是 GET，则显示表单，并且将用户已有信息也显示在其中；如果是 POST，则接收用户提交的表单信息，然后更新各个数据模型实例属性的值，最后不要忘记保存。

编写完视图函数后再编写相应的模板代码，在./templates/account 目录中创建文件 myself_edit.html，其代码如下。

```
{% extends "base.html" %}
{% block title %}my information{% endblock %}
{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>Edit My Information</h1>
    <div class="row">
        <div class="col-md-6" >
        <form class="form-horizontal" action="." method="post">{% csrf_token %}
            <div class="row">
                <div class="col-md-4 text-right"><span>用户名:</span></div>
                <div class="col-md-8 text-left">{{user.username}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>邮箱:</span></div>
                <div class="col-md-8 text-left">{{user_form.email}}</div>
            </div>
            <div class="row">
```

```
                <div class="col-md-4 text-right"><span>生日:</span></div>
                <div class="col-md-8 text-left">{{userprofile_form.birth}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>电话:</span></div>
                <div class="col-md-8 text-left">{{userprofile_form.phone}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>毕业学校:</span></div>
                <div class="col-md-8 text-left">{{userinfo_form.school}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>工作单位:</span></div>
                <div class="col-md-8 text-left">{{userinfo_form.company}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>职业:</span></div>
                <div class="col-md-8 text-left">{{userinfo_form.profession}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>地址:</span></div>
                <div class="col-md-8 text-left">{{userinfo_form.address}}</div>
            </div>
            <div class="row">
                <div class="col-md-4 text-right"><span>个人介绍:</span></div>
                <div class="col-md-8 text-left">{{userinfo_form.aboutme}}</div>
            </div>
            <div class="row">
                <input type="submit" class="btn btn-primary btn-lg" value="Submit">
            </div>
        </form>
        </div>
        <div class="col-md-6">
            <p>picture</p>
        </div>
    </div>
</div>
{% endblock %}
```

这段代码没有引入新的知识，读者应该能够理解，如果在理解上有困难，可以查阅前文相关知识，或者通过本书所公布的联系方式通过网络与笔者讨论。当然，不要忽视笔者推荐的解决问题的方式——Google。

万事俱备，就差配置 URL 了，在./account/urls.py 中增加一行代码。

```
path('edit-my-information/', views.myself_edit, name="edit_my_information"),
```

运行 Django 服务，并实现一个账户登录之后，在浏览器的地址栏中输入 http://localhost:8000/account/edit-my-information/，如果此前没有完善该用户的信息，界面如图 2-43 所示。

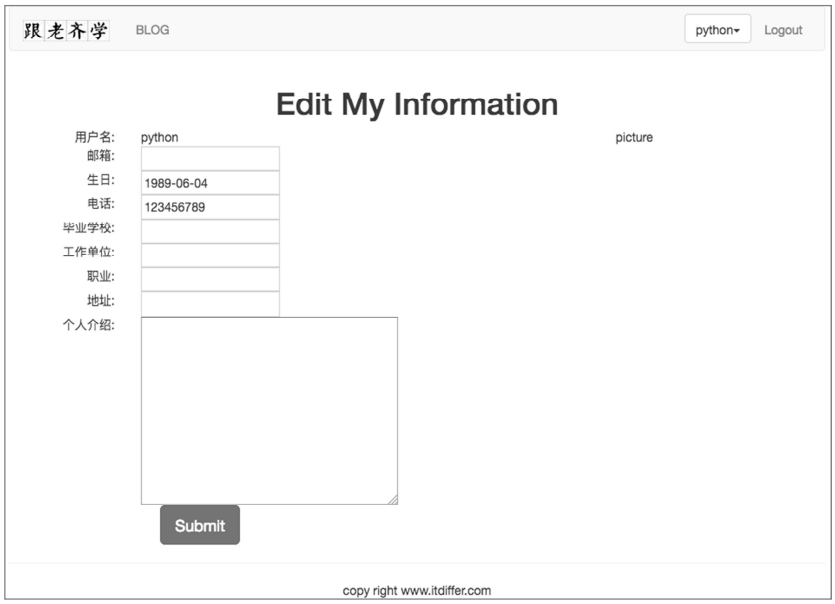

图 2-43　个人信息编辑页面

按照要求，完善个人信息，然后单击 Submit 按钮，会跳转到个人信息展示界面，如图 2-44 所示。

图 2-44　个人信息页面

原来没有完善的个人信息都被完善了。

为了让修改和查看个人信息更方便，需要提供上述操作的入口。是否还记得我们曾经在导航栏的用户名那里做了一个下拉菜单，作为“修改密码”的入口，下面仿照这种操作，做一个查看自己信息的入口。

编辑./templates/header.html 文件，找到“修改密码”那一行，在其下增加“个人信息”的下拉菜单代码。

```
<li><a href="{% url 'account:my_information' %}">个人信息</a></li>
```

保存后刷新页面，其效果如图 2-45 所示。

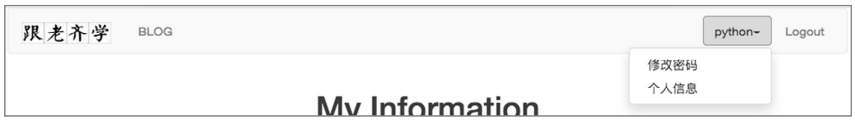

图 2-45　个人信息入口

再编辑./templates/account/myself.html 文件，在所展示的信息下面增加一个按钮，通过这个按钮实现编辑功能，在个人信息列表内容下面增加如下代码。

```
<a href="{% url 'account:edit_my_information' %}">
    <button class="btn btn-primary btn-lg">edit my information</button>
</a>
```

保存后刷新页面，其效果如图 2-46 所示。

图 2-46　增加编辑按钮的个人信息页

单击 edit my information 按钮，就能够实现编辑当前用户信息的功能。

2.6.4　上传和裁剪头像图片

在展示和编辑个人信息的过程中，细心的读者是否看到界面的右侧有一个地方写有“picture”字样，但是什么也没有做，仅仅占据了一个 div 的位置罢了。现在就把它完善，打算在那里上传用户的头像图片。

为了显示那里是头像，先传一张图片上去。在./static/images 目录中放一张图片，下面的代码中会用到，笔者放了一张名为 newton.jpg 的图片。编辑./templates/account/myself.html 文件，找到写有 pictures 的代码块，将原来的 picture 部分用下面的代码替换。

```
{% load staticfiles %}    #①
<div style="margin-right:100px">
    <img name="user_face" src="{% static 'images/newton.jpg' %}"
         class="img-circle" width="270px" id="my_photo">
</div>     #②
<div style="margin-right:100px">
    <button class="btn btn-primary btn-lg" id= "upload_image"
            onclick="upload_image_layer()">upload my photo
    </button>
</div>    #③
```

因为要引用静态文件，所以语句①是必须有的。语句②（ 标签）用于显示图片。语句③（ 标签）用于显示一个按钮，不过这个按钮预留了一个操作“onclick= "upload_image_layer()"”，后面会编写这个函数。刷新界面后，就可以看到作为头像的图片，如图 2-47 所示，只不过这还不是上传的。

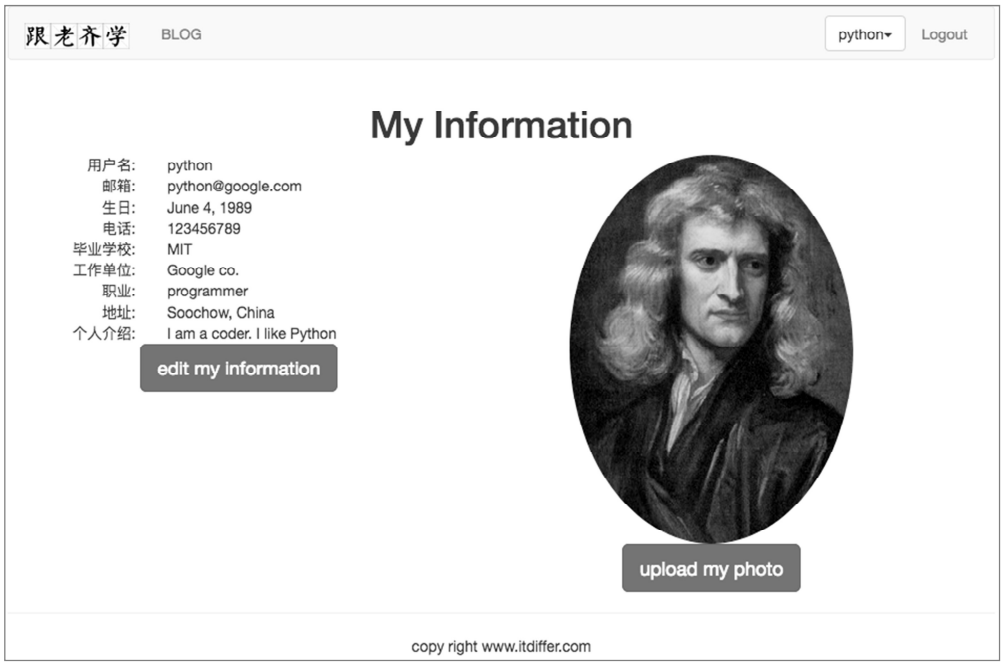

图 2-47　显示个人头像

占领位置不是目的，目的是实现图片上传。

1．部署上传和裁剪头像图片的插件

很多网站都有头像图片上传和裁剪头像图片的功能，这里使用实现这个功能的插件（是笔者在网上找到的），完成头像图片的上传和展示。

读者可以到 https://github.com/qiwsir/DjangoPracticeProject 下载 ImgCrop 文件包。当然，如果找到了别的类似功能插件也可以使用。这里姑且以此为例来说明使用方法。

下载文件包后，观察其目录和文件结构，如下所示。

```
/ImgCrop$ tree
.
├── css
│   └── style.css
├── images
│   └── avatar.png
├── index.html
└── js
    ├── cropbox.js
    ├── cropbox-min.js
└── jquery-1.11.1.min.js
```

从上面的文件结构中可以看出，这是一个纯粹前端的工具，那么就把各个文件按照我们已经部署好的项目架构分门别类地放置。

- 复制 ImgCrop/css/style.css 到本项目的./static/css 中，并更名为 imagecrop.css。为了达到后期的展示效果，将文件中的样式修改一下，目的是将显示区域左对齐。

```
.container {
    width: 400px;
    margin: 0 auto 0 0;
    position: relative;
    font-family: 微软雅黑;
    font-size: 12px;
}
```

- 复制 ImgCrop/jquery-1.11.1.min.js 到本项目的./static/js 中，虽然原来已经有了一个 jQuery.js 文件，但还要复制这个文件，因为版本不同。因为所选用的 JavaScript 插件比较古老，所以使用这个版本的 jQuery。
- 复制 ImgCrop/js/cropbox-min.js 到本项目的./static/js 中。

完成上述文件的复制后，开始在本项目中实现头像图片的裁剪。

先在./account/views.py 中增加一个非常简单的视图函数，代码如下。

```
def my_image(request):
    return render(request, 'account/imagecrop.html',)
```

然后在./templates/account 目录中创建 imagecrop.html 文件，并且将裁剪头像插件文件夹中的 index.html 文件里面的部分代码复制到 imagecrop.html 文件中，同时增加 CSS 和 JavaScript 文件，最终代码如下。

```
{% load staticfiles %}
<link rel="stylesheet" href="{% static 'css/imagecrop.css' %}">
<div class="container">
  <div class="imageBox">
    <div class="thumbBox"></div>
    <div class="spinner" style="display: none"></div>
  </div>
  <div class="action">
    <!-- <input type="file" id="file" style=" width: 200px">-->
    <div class="new-contentarea tc"> <a href="javascript:void(0)" class="upload-img">
      <label for="upload-file">请先选择图片...</label>
      </a>
      <input type="file" class="" name="upload-file" id="upload-file" />
    </div>
    <input type="button" id="btnCrop" class="Btnsty_peyton" value="OK">
    <input type="button" id="btnZoomIn" class="Btnsty_peyton" value="+"  >
    <input type="button" id="btnZoomOut" class="Btnsty_peyton" value="-" >
  </div>
  <div class="cropped"></div>
</div>

<script src="{% static 'js/jquery-1.11.1.min.js' %}"></script>
<script type="text/javascript" src="{% static 'js/cropbox-min.js' %}"></script>
```

```
<script type="text/javascript">

$(window).load(function() {
    var options =
    {
        thumbBox: '.thumbBox',
        spinner: '.spinner',
        imgSrc: ''
    }
    var cropper = $('.imageBox').cropbox(options);
    var img="";
    $('#upload-file').on('change', function(){
        var reader = new FileReader();
        reader.onload = function(e) {
            options.imgSrc = e.target.result;
            cropper = $('.imageBox').cropbox(options);
        }
        reader.readAsDataURL(this.files[0]);
        //getImg();
    })
    $('#btnCrop').on('click', function(){
        alert("图片上传喽");
    })
    function getImg(){
        img = cropper.getDataURL();
        $('.cropped').html('');
        $('.cropped').append('<img src="'+img+'" align="absmiddle"
                                    style="width:180px; margin-top:4px;
                                           border-radius:180px;
                                           box-shadow:0px 0px 12px #7E7E7E;">
                                <p>180px*180px</p>');
        $('.cropped').append('<img src="'+img+'" align="absmiddle"
                                    style="width:128px; margin-top:4px;
                                           border-radius:128px;
                                           box-shadow:0px 0px 12px #7E7E7E;">
                                <p>128px*128px</p>');
        $('.cropped').append('<img src="'+img+'" align="absmiddle"
                                    style="width:64px; margin-top:4px;
                                           border-radius:64px;
                                           box-shadow:0px 0px 12px #7E7E7E;" >
                                <p>64px*64px</p>');
        }

    $(".imageBox").on("mouseup",function(){
        getImg();
        });

    $('#btnZoomIn').on('click', function(){
        cropper.zoomIn();
    })
    $('#btnZoomOut').on('click', function(){
```

```
        cropper.zoomOut();
    })
});
</script>
```

这么多代码，没有几行是自己写的，都是从那个插件的 index.html 文件中复制过来的。为了显示真的是复制，原来文件中注释掉的内容笔者都没有删除，读者可以对照看一下。

特别提醒的是开头引入的 CSS 和 JavaScript 文件，不要把路径搞错。还有一个需要注意的地方，就是<input type="button" id="btnCrop" class="Btnsty_peyton" value="OK">这行代码，原文件中没有增加 id，如果读者直接复制过来，在操作的时候单击 OK 按钮，可能会没有显示。

建议读者将此代码分析一下，理解大概的执行过程。

最后在./account/urls.py 中配置 URL，增加下面一行代码。

```
path('my-image/', views.my_image, name="my_image"),
```

重新启动 Django 服务，在浏览器的地址栏中输入 http://localhost:8000/account/my-image/，会看到如图 2-48 所示的界面。

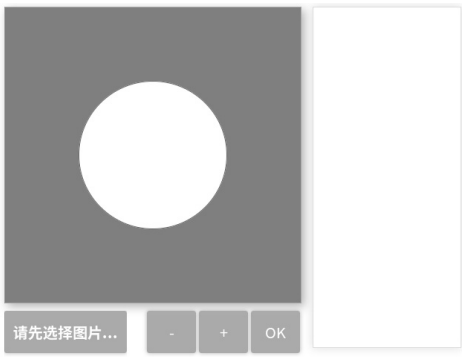

图 2-48　上传个人头像界面

单击“请先选择图片”按钮，选择一张图片，看一下效果（见图 2-49）。单击 OK 按钮后，该图片还不能上传到服务器上，只是弹出一个提示语。

图 2-49　上传图片的效果

为了实现图片的真正上传，还要做不少工作，下面逐项完成。

2．修改数据模型

上传的文件也要存储在数据库中，否则无法显示。前面曾经建立了一个 UserInfo 的用户个人信息模型类，但是当时没有考虑上传头像问题，现在需要在那个类中增加一个字段。

编辑./account/models.py 文件，在 UserInfo 类中增加如下字段。

```
photo = models.ImageField(blank=True)
```

在 ImageField(blank=True)中设置参数 blank=True，意为用户的头像可以为空。

每次完成 models.py 的编辑，都要重新做迁移数据的操作，依次执行下述指令。

```
$ python3 manage.py makemigrations
$ python3 manage.py migrate
```

数据模型修改好之后，不要忘记./account/forms.py 中的 UserInfoForm 表单类也要在 fields 的值中增加 photo，如下所示。

```
fields = ("school", "company", "profession", "address", "aboutme", "photo")
```

3．编写视图函数

下面所编写的视图函数专门用于处理显示、上传和裁剪头像。前面编写了 my_image()视图函数，主要是为了显示上传和裁剪的效果，现在要对这个函数进行修改，从而实现存储前端传过来的图片的功能。

```
@login_required(login_url='/account/login/')
def my_image(request):
    if request.method == 'POST':
        img = request.POST['img']    #①
        userinfo = UserInfo.objects.get(user=request.user.id)    #②
        userinfo.photo = img
        userinfo.save()
        return HttpResponse("1")
    else:
        return render(request, 'account/imagecrop.html',)
```

语句①得到前端以 POST 方式提交的图片信息，这里已经规定前端提交的类字典数据中有 img 这个键，否则会报错。其他代码都是比较熟悉的内容，如果读者有不理解的地方，可以查阅本书前面的相关内容。

在调试的时候，如果读者使用的登录账号是在 2.6 节以前生成的，会在语句②处报错（原因请查阅 2.6 节有关叙述）。那么，语句②其实可以再次优化，不过这个优化的任务就交给读者完成了，可以参考同一个文件中的其他视图函数写法。

4．再次编写前端代码

之所以再写，是因为前面对上传和裁剪功能已经创建完成，但是，还没有把数据提交给后端的视图函数。

下面检查刚才已经做好的上传和裁剪头像功能，上传一张图片，然后单击 OK 按钮，会弹出一个提示框，如图 2-50 所示。

图 2-50　上传测试图片

显然，我们需要的功能不是弹出一个提示框，而是将裁剪的图片上传。为了实现这个功能，这里使用一种新的从前端向后端提交数据的方式，也是在网站开发中被普遍使用的 Ajax。什么是 Ajax？推荐读者阅读维基百科的说明（https://en.wikipedia.org/wiki/Ajax_(programming)），此处只介绍实现方法。

编辑./templates/account/imagecrop.html 文件中的 JavaScript 代码，修改单击 OK 按钮后实现的功能，代码如下。

```
$('#btnCrop').on('click', function(){
    //alert("图片上传喽");
    $.ajax({
        url: '{% url "account:my_image" %}',
        type: 'POST',
        data: {"img": img},
        success: function(e){
            location.href= "{% url 'account:my_information' %}"
        },
    });
})
```

这段代码就是 Ajax 的一种写法（使用 jQuery），请读者仔细观察其结构。

- url：声明提交地址。
- type：声明提交方式，有 GET 和 POST 两种，此处使用 POST。
- data：声明提交的数据内容。
- success：提交成功后，根据反馈结果实现页面跳转。

关于 jQuery 的 Ajax 方法的参数，这里仅使用上述几种，更多的参数请读者阅读 http://www.w3school.com.cn/jquery/ajax_ajax.asp 中的内容。

保存文件后刷新界面，选择一张图片，单击 OK 按钮上传图片。

不要激动！

这里应该报错，也一定要报错，只有报错才能显示 Django 的特点，如图 2-51 所示。

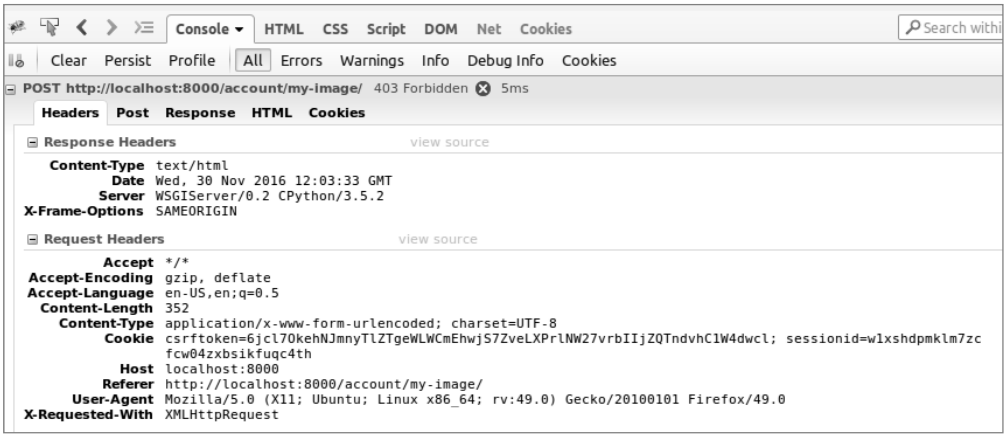

图 2-51　上传图片报错

这个报错是什么意思？

现在要进行的操作是最重要的。打开 Google，输入“Django 403 forbidden”，会看到很多讨论这个问题的内容，都在说是与 CSRF 有关。不要忘记，前面已经提到过这个问题，在没有使用 Ajax 提交数据时，都是在表单代码中增加{% csrf_token %}，才能实现数据的提交。现在使用 Ajax 方式提交，没有使用增加代码的方式，也没有使用别的替代方法，所以 Django 从安全角度考虑，就拒绝了提交数据。

解决方法有很多，网上列出了好多种。这里使用其中一种，将一个名为 csrf.js 的文件引入进来，这个文件读者可以在 https://github.com/qiwsir/DjangoPracticeProject/tree/master/mysite/static/js/csrf.js 下载，将这个文件放在本项目的./static/js 目录中，并且在./templates/account/imagecrop.html 里面引用。

```
<!—其他代码-->
</div>
<script type="text/javascript" src="{% static 'js/csrf.js' %}"></script>
<script type="text/javascript">
<!—后续代码-->
```

保存文件后刷新页面，上传一张图片，单击 OK 按钮，发现界面跳转到指定页面了。

看来图片应该被上传了。还是老办法，看看数据库中保存的内容是否发生了变化。

```
sqlite> select user_id, photo from account_userinfo;
user_id     photo
----------  ----------------------------------------------------------------
8           ……
sqlite> select username from auth_user where id=8;
username
----------
python
```

从上述数据库查询结果可以看出，当前用户的头像已经上传并以 base64 格式保存到了数据库表 account_userinfo 的字段 photo 下面。注意，并没有在服务器某个位置保存图片文件（类似读者在自己的计算机上看到的图片那样），而是通过所使用的那个插件，将实体图片转换为代码格式，保存到了数据库中。

但是，在显示图片的位置还没有将刚刚上传的图片显示出来。所以还要修改./template/account/myself.html 文件，对显示图片的部分代码进行如下修改。

```
<div class="col-md-6">
<div style="margin-right:100px">
       {% if userinfo.photo %}
       <img src="{{ userinfo.photo | striptags }}"
            class="img-circle" id="my_photo" name="user_face">    #①
       {% else %}
       <img name="user_face" src="{% static 'images/newton.jpg' %}"
            class="img-circle" id="my_photo">
       {% endif %}
    </div>
    <div style="margin-right:100px"><button class="btn btn-primary btn-lg" id="upload_image"
onclick="upload_image_layer()">upload my photo</button></div>
</div>
```

这段代码对 userinfo.photo 进行判断，如果用户还没有上传图片，则显示默认的 newton.jpg；如果用户已经上传图片了，则显示上传的图片内容。

在显示上传图片内容时，语句①使用了{{ userinfo.photo | striptags }}这种方式，这是 Django 模板中常用的一种方式。“|”符号是管道符，是将 userinfo.photo 这个从数据库中读出来的对象内容进行过滤，按照什么规则过滤呢？要看管道符后面的内容了，striptags 的意思是去除 HTML 标签，就是将 userinfo.photo 所引用的对象内容中的 HTML 标签去掉，然后将结果返回。

再刷新一下刚刚跳转过来的界面，比刚才漂亮了，呈现了一张头像，如图 2-52 所示。

图 2-52　个人头像

这样就实现了用户头像图片的裁剪功能。

但是，这样上传图像还不友好，还需要继续优化。

2.6.5　优化头像上传功能

头像上传和裁剪虽然实现了，但是使用该功能的入口还没有做好，所以还要继续优化。

入口就是单击 upload my photo 按钮之后，会弹出一个提示框，在这个提示框中完成上传和裁剪图片的功能。

提示框是 Web 开发中常用的功能，所以一定有现成的工具可以使用，笔者在这里使用来自 http://layer.layui.com/网站的提示框。下载后，根据本项目的静态文件部署规范，将提示框各个文件放到相应的目录。如果读者在阅读本书时，上述网站因为某种原因无法访问，还可以到 https://github.com/qiwsir/DjangoPracticeProject 下载提示框。

具体部署方法就是将 layer.js 文件和 skin 目录复制到本项目的./static/js 目录里面，然后编辑./templates/account/myself.html 文件，在{% endblock %}前面用下面的方式将插件引入。

```
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
```

接着，写下一段简单的 JavaScript 代码，先看看效果。

```
<script>
function upload_image_layer(){
    layer.open({
        content:"<p>hello world</p>",
    });
}
</script>
```

检查 Django 服务是否已经运行，在 http://localhost:8000/account/my-information/页面中，如果是原来打开的，就刷新一下，单击 upload my photo 按钮，会弹出一个提示框，如图 2-53 所示。

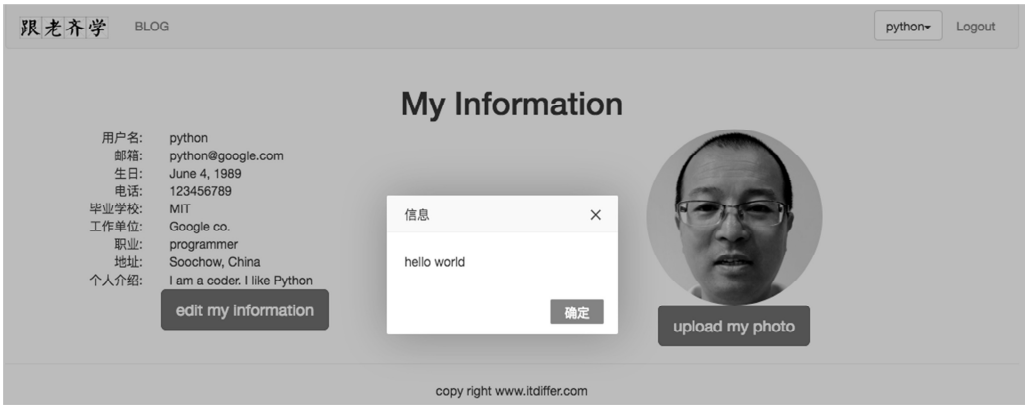

图 2-53　测试弹出提示框

这就是我们想要的效果。下面的任务就是让系统弹出上传和裁剪头像的层。

将刚才的测试代码修改为下面的代码。

```
<script>
function upload_image_layer(){
    layer.open({
        title:"上传头像",
        area: ['650px', '600px'],
        type:2,
        content:"{% url 'account:my_image' %}",
    });
}
</script>
```

刷新页面之后，看看效果，应该弹出一个相当不错的提示框，如图 2-54 所示。

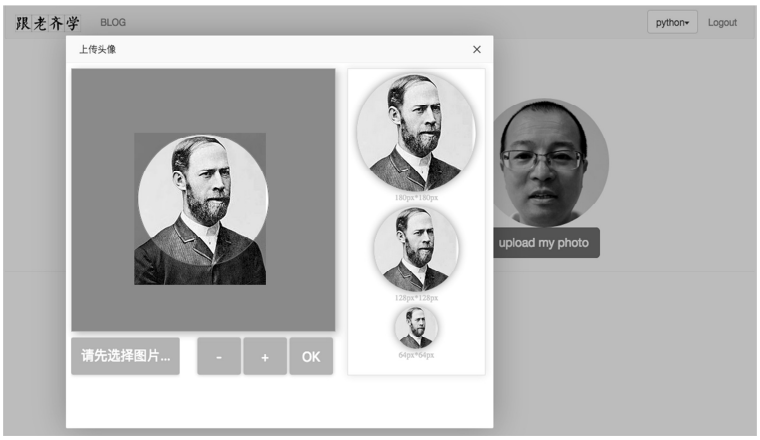

图 2-54　在弹出提示框中上传头像

这时候，单击提示框中的 OK 按钮，就能将图片上传，但是会遇到下一个麻烦，不信你试试。

看到那个麻烦了吗？这可不是因为笔者使用了大胡子的照片而引起的麻烦，即使换成其他照片也一样会出现麻烦。读者不要觉得可笑，我们很多时候都在运用这种“掩耳盗铃”的思维方式。比如，当你辛辛苦苦做的网站发布了，在同事那里访问都没有问题，偏偏老板说他访问不了，因此他认为你做得有问题。

不要掩耳盗铃，要以科学严谨的态度查找原因。我们看到麻烦的来源在./template/account/imagecrop.html 中，在图片上传后，直接转到了 http://localhost:8000/account/my-information/，以前这么做可以，感觉还不错，但是现在因为将 imagecropt.html 页面嵌入了弹出的提示框，所以就不能再如从前一样跳转了。我们想要实现的是上传之后自动关闭提示框，并刷新提示框的父页面，在该页面中显示所上传的头像，因此要对./template/account/imagecrop.html 中的 JavaScript 代码进行一些改写。

```
$('#btnCrop').on('click', function(){
//alert("图片上传喽");
    $.ajax({
        url: '{% url "account:my_image" %}',
        type: 'POST',
        data: {"img": img},
        success: function(e){
            //location.href= "{% url 'account:my_information' %}"
          //下面是增加的代码
          ////相应的视图函数中的 return HttpResponse("1"), 其数值在这里被应用
            if(e == "1"){
                parent.location.reload();
            } else {
                alert("sorry, you are not lucky. the picutre can't been uploaded.")
            }
        },
    });
})
```

再次刷新 http://localhost:8000/account/my-information/，上传头像图片，看一下效果，已经能够关闭提示框并刷新页面了。

2.6.6 对个人信息进行管理

用 admin 账号登录 http://localhost:8000/admin/，在管理后台中，可以对 Users 和 User profiles 这两类个人信息进行管理，但是新增加的 User infos 还没有被列出来，如图 2-55 所示。

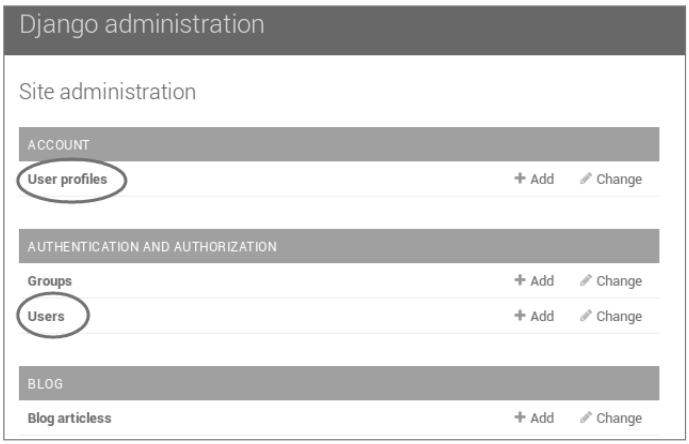

图 2-55 后台管理界面

要增加 User infos，需要再次编辑./account/admin.py 文件，增加如下代码。

```
from .models import UserInfo
class UserInfoAdmin(admin.ModelAdmin):
    list_display = ("user", 'school', 'company', 'profession',
                    'address', 'aboutme', 'photo')
    list_filter = ("school", "company", "profession")

admin.site.register(UserInfo, UserInfoAdmin)
```

注意，在文件开头要引入 UserInfo 类。增加代码之后，再次刷新上述页面，效果如图 2-56 所示。

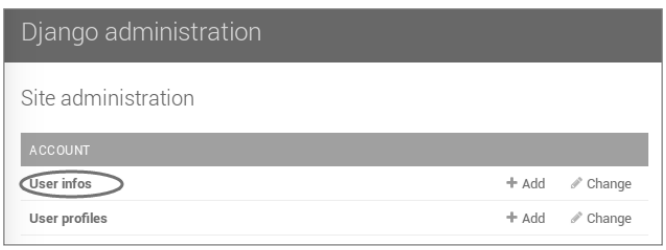

图 2-56 增加了 User infos 管理项

单击 User infos 选项，可以查看所有用户的相关个人信息，如图 2-57 所示。

图 2-57　管理界面的个人信息

虽然界面美观度稍差，但功能是齐全的，也可以在这里完善某个用户的个人信息。

回顾本节的学习内容，主要是前端知识，即 HTML/CSS/JS 部分，且以 JavaScript 为主。对于一个网站来说，后端固然重要，但是前端是跟用户打交道的，要提高其友好性，就不得不大量使用 JavaScript 脚本或者组件（插件）。作为一名立志从事 Web 开发的程序员，除学习 Django 外，前端知识也不能或缺，虽然本书没有专门讲解，但在项目中也要大量应用（最基本的）。

2.6.7　知识点

1．Ajax

Ajax 的全称是 Asynchronous JavaScript and XML，译为“异步 JavaScript 和 XML”。为什么要异步？在互联网的“石器”时代，网页都是 HTML 的静态页面，假设两个页面只有很小的差别，也必须完全刷新，才能访问另一个页面。另外，在向服务器提交网页端表单时，也有类似重复提交的情况。为此，就需要有某种异步的技术，实现局部信息请求，比如曾经的 Java applets——目前已经不用的技术。Google 公司在一系列的项目中使用了 Ajax 技术，并让这种技术变得流行。在现在网站开发中，Ajax 技术已经被广泛使用，所以读者必须要了解。本书中对此技术的应用实例还是初级阶段的，建议读者查阅有关图书和资料，深入学习。

2．模型：主键

在关系型数据库中，“主键”是常见的概念，它的作用就是保证整个数据库表中记录的唯一性。在 Django 的数据模型类中，虽然我们没有明确规定哪个字段是主键，但因为主键是必需的，所以 Django 默认在每个数据模型类中都隐藏一个名为 id 的字段，并且这个字段是 AutoField 类型（自增整数型）。在完成数据迁移后，数据库表中都会有一个 id 字段，它就是这个表的主键。

在某些情况下，或许要自定义主键，方法就是在数据模型类的某个字段的属性参数中声明 primary_key = True，这样该字段就被指定为主键了，而那个没有显示出来的默认主键就被忽略了。

一个数据模型类中只能有一个字段被指定为主键。请不要把允许有重复的字段作为主键。

3．模型：唯一性

在数据模型类中，如果要确定某个字段的值不能重复、是唯一的，那么可以在该字段的属性参数中声明 unique=True。注意，唯一不等于该字段就是主键。

4．模型：一对一

定义数据库表之间的关系是关系型数据库中非常重要的操作，Django 对此功能的实现是通

过在数据模型类中定义字段属性来实现的。比如，在本节的 UserInfo 数据模型类中就定义了 user = models.OneToOneField(User, unique=True)，UserInfo 与 User 类之间是一对一的关系，每个用户对应着一个关于他（她）的拓展信息表。官方文档中也有对此关系的详细描述，请读者自行阅读。

5．文档导读

（1）*One-to-one relationships*，https://docs.djangoproject.com/en/2.1/topics/db/examples/one_to_one/。

（2）*django-ajax*，https://github.com/yceruto/django-ajax。

（3）*How do I integrate Ajax with Django applications*，http://stackoverflow.com/questions/20306981/how-do-i-integrate-ajax-with-django-applications。

第3章

文章管理和展示

网站中必须有内容，如果按照内容产生的方式来分类，就目前来讲可以分为两类，一类是“用户生成内容（UGC）”，比如“YouTube”“Twitter”；另一类是“专业生产内容（PGC/PPC）”，比如“跟老齐学 itdiffer.com”。

本章要学习的项目是做“用户生成内容”的网站。这种网站有前后两部分，“前面”是让访客浏览网站上的文档，“后面”是用户管理自己的文章。

3.1 管理文章栏目

每个发布文章的用户都希望能用“栏目”来对自己的文章进行归类，不至于让自己的页面显得凌乱，如图 3-1 所示。

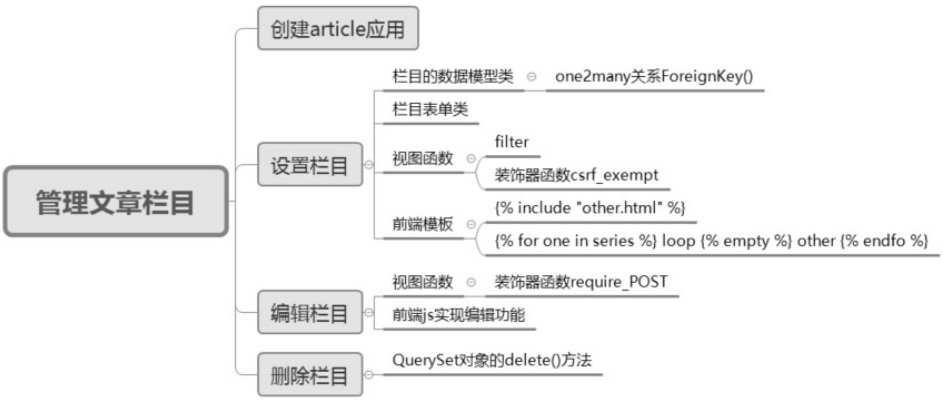

图 3-1　知识技能导图

3.1.1 设置栏目

在实现对文章的管理功能之前，要创建“文章管理”的应用，这是必需的，以区别前面的各种应用。

```
$ python3 manage.py startapp article
```

首先创建一个名为 article 的应用，并且要在./mysite/settiing.py 中进行配置。

```
INSTALLED_APPS = [
#…其他应用
    'article',
]
```

然后就是我们已经熟悉的步骤了，创建数据模型、表单、视图函数、前端模板和配置 URL。这一系列操作完成后，基本上的功能目标就实现了，看似简单，但真正实施起来，还是有不少需要小心的地方或者需要补充的知识技能。

1．栏目的数据模型

对于文章栏目，这里不做多级栏目设置，只设置一级栏目（笔者不建议让用户一层一层地单击才能找到想要的东西），所以相对来讲，栏目的数据模型稍微简单一些。编辑./article/models.py 文件，输入如下代码。

```
from django.db import models
from django.contrib.auth.models import User

class ArticleColumn(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='article_column')    #①
    column = models.CharField(max_length=200)
    created = models.DateField(auto_now_add=True)

    def __str__(self):
        return self.column
```

用户和文章栏目之间是“一对多”的关系，即一个用户可以设置多个文章栏目，通过语句①实现 ArticleColumn 和 User 之间的这种关系。在 Django 中，模型对象之间的关系可以概括为“一对一”、“一对多”和“多对多”三种关系，分别对应 OneToOneField、ForeignKey、ManyToManyField。读者在此暂时不用纠缠三者的深刻内涵，先按照本书中所提供的代码操作，等到后续知识增加了，自然知晓。

数据模型建好后就要迁移数据了，生成数据库表（请读者自行完成此操作，如果没有掌握迁移数据的方法，请参阅本书前面的叙述）。

有了数据模型，自然少不了表单类，因为要通过表单填写栏目名称，即为 column 字段赋值。所以，要创建./article/forms.py 文件，并编写如下代码。

```
from django import forms
from .models import ArticleColumn

class ArticleColumnForm(forms.ModelForm):
    class Meta:
        model = ArticleColumn
        fields = ("column",)
```

基本结构的内容暂告一段落，下面开始做应用部分。

2．简易视图函数

为了能够对即将做的东西有一个直观的感受，我们暂时不写表单类，而是写一个简单的视图函数，先看看想要达到的效果。

编辑./article/views.py 文件，输入如下代码。

```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from .models import ArticleColumn

@login_required(login_url='/account/login/')
def article_column(request):
    columns = ArticleColumn.objects.filter(user=request.user)    #②
    return render(request, "article/column/article_column.html", {"columns":columns})
```

在 article_column()函数中，主要是通过语句②将数据库表中该用户所属的栏目都读取出来。语句②本质上是两条语句合并起来的，一条是 ArticleColumn.objects.all()，然后根据 user=request.user 的条件进行筛选，即 ArticleColumn.objects.filter(user=request.user)，这两个查询行为可以用语句②这样的一条指令来表达。

接下来配置./mysite/urls.py 中的 URL。

```
path('article/', include('article.urls', namespace='article')),
```

再创建./article/urls.py 文件，设置本应用的 URL。

```
from django.urls import path
from . import views

app_name = "article"

urlpatterns = [
    path('article-column/', views.article_column, name="article_column"),
]
```

下面就要编写前端模板了。一切都是按照规定的步骤在有条不紊地进行着。

3．前端模板

现在要实现用户管理自己的文章栏目，对于这种管理功能，笔者把它视为所谓的“后台”，即不是显示给所有用户看的，只有发布内容的用户才能使用。所以，显示样式上也有所变化。

在./templates 目录中建立 article 子目录，然后创建./templates/article/header.html 文件，其代码如下。

```
{% load staticfiles %}
<div class="container">
    <nav class="navbar navbar-default" role="navigation">
        <div class="navbar-header">
            <a class="navbar-brand" href="http://www.itdiffer.com">
                <img width="100px" src="{% static 'images/backlogo.png' %}">
            </a>
        </div>
```

```
        <div>
            <ul class="nav navbar-nav" role="tablist">
                <li><a href="{% url 'article:article_column' %}">文章管理</a></li>
            </ul>
            <ul class="nav navbar-nav navbar-right" style="margin-right:10px">
                <li><a href="{% url 'blog:blog_title' %}">网站首页</a></li>
                <li><span>{{ user.username }}</li>
                <li><a href="{% url 'account:user_logout' %}">Logout</a></li>
            </ul>
        </div>
    </nav>
</div>
```

读者仔细观察上述代码，能够发现与./templates/header.html 源码相差不大。

读者在这里应该能够想象一个过程，在用户登录后，要管理自己的文章栏目，就要进入所谓的“后台”，可是入口在哪里？所以，也要为登录用户设置入口，入口位置与前面设置的“修改密码”和“个人信息”的位置一样，所以顺便修改./templates/header.html 文件设置入口。注意，这里通过入口进入后台的页面，笔者选择了现在正准备做的这个页面，当然也可以修改为其他页面。

```
<li><a href="{% url 'article:article_column' %}">后台管理</a></li>
```

上述代码，读者能够找到适合的位置安放吧。一定能！我确信。

后台部分的 footer.html 文件，可以继续使用./templates/footer.html。

一般的管理后台，左侧都有一个功能栏目，我们也来做一个。创建./templates/article/leftslider.html 文件，代码如下。

```
<div class="bg-info">
    <div class="text-center" style="margin-top: 5px;">
        <p><a href="{% url 'article:article_column'%}">栏目管理</a></p>
    </div>
</div>
```

下面就组装./templates/article/base.html 文件，代码如下。

```
{% load staticfiles %}
<!DOCTYPE html>
<html lang="zh-cn">
<head>
    <meta http-equiv="X-UA-Compatible" content="IE=Edge">
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'css/bootstrap.css' %}">
</head>
<body>
<div class="container">
    {% include 'article/header.html' %}
    <div class="col-md-2">
    {% include 'article/leftslider.html' %}
    </div>
```

```
    <div class="col-md-10">
    {% block content %}
    {% endblock %}
    </div>
    {% include 'footer.html' %}
</div>
</body>
</html>
```

上面做的都是准备工作，还没有编写视图函数所要求的模板呢！下面就建立./templates/article/column/article_column.html 文件，并且输入如下代码。注意，在./templates/article/中要建立目录 column。

```
{% extends "article/base.html" %}
{% load staticfiles %}
{% block title %}article column{% endblock %}
{% block content %}
<div>
    <p class="text-right"><button class="btn btn-primary">add column</button></p>
    <table class="table table-hover">
     <tr>
          <td>序号</td>
          <td>栏目名称</td>
          <td>操作</td>
     </tr>
     {% for column in columns %}
     <tr>
          <td>{{ forloop.counter }}</td>
          <td>{{ column.column }}</td>
          <td>--</td>
     </tr>
     {% empty %}
     <p>还没有设置栏目，太懒了。</p>
     {% endfor %}
    </table>
</div>
{% endblock %}
```

以{% for column in columns %}实现表格中每行数据的输出。{{ forloop.counter }}中的 forloop 只在循环内部起作用，它是一个模板变量，具有提示循环进度的属性，比如这里使用 forloop.counter 的效果是得到每个循环的顺序号，其本质是显示当前循环次数的计数器（所以从 1 开始了）。如果变量 columns 引用的对象为空，则通过{% empty %}执行后面的内容。循环语句中的{% empty %}省略了通过 if 来判断。

模板做好后，运行 Django 服务，在浏览器的地址栏中输入 http://localhost:8000/article/article-column/，如果用户没有登录，会跳转到登录界面，之后看到如图 3-2 所示的效果。

图 3-2　文章管理的列表页

目前这个用户还没有设置任何栏目，所以是上面的显示效果。

下一步的目标现在就明确了，要增加新栏目。

4．增加新栏目

增加新栏目的操作流程是单击 add column 按钮，弹出一个对话框，在这个对话框中输入新的栏目名称。

为了实现弹出对话框，还是用第 2 章中使用的 layer.js 插件。编辑./templates/article/column/article_column.html 文件，将原来的<button>进行适当修改，代码如下。

```
<button id= "add_column" onclick="add_column()" class="btn btn-primary">
    add column
</button>
```

当用户单击这个按钮时，触发 add_column()函数（JavaScript 函数），这个函数代码如下（下面的代码放在文件尾部{% endblock %}之内）。提醒读者，这种将 JavaScript 函数和 HTML 绑定到一起写的方法，在真实的项目中不值得提倡，本书中的项目因为主要是学习 Django，所以为了阅读和理解方便，就没有将代码分开，请读者注意。

```
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
<script type="text/javascript">
    function add_column(){
        var index = layer.open({
            type:1,
            skin:"layui-layer-rim",
            area: ["400px", "200px"],
            title: "新增栏目",
            content: '<div class="text-center" style="margin-top:20px">
                        <p>请输入新的栏目名称</p>
                        <p><input type=”text”></p>
                    </div>',
            btn:['确定', '取消'],
            yes: function(index, layero){    #④
                column_name = $('#id_column').val();
                alert(column_name);
            },
            btn2: function(index, layero){    #⑤
                layer.close(index);
```

```
            }
        });
    }
</script>
```

在增加的代码中，引入 jQuery 和 layer 是必需的。重点观察 add_column()函数，在弹出的对话框中，语句④是针对第一个“确定”按钮的，这里我们仅仅获取写入的栏目名称。语句⑤是针对第二个“取消”按钮的。

先看看效果，尽管只是前端，还没有将数据向后端发送。单击 add column 按钮，效果如图 3-3 所示。

图 3-3　测试弹出提示框效果

这仅仅是效果，要实现真正的业务，还要继续完善后端的代码。

编辑./article/views.py 文件，重写 article_column()视图函数，下面是所有代码。

```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse

from .models import ArticleColumn
from .forms import ArticleColumnForm

@login_required(login_url='/account/login/')
@csrf_exempt    #⑥
def article_column(request):
    if request.method == "GET":
        columns = ArticleColumn.objects.filter(user=request.user)
        column_form = ArticleColumnForm()
        return render(request, "article/column/article_column.html",
                      {"columns":columns, 'column_form':column_form})

    if request.method == "POST":
        column_name = request.POST['column']
        columns = ArticleColumn.objects.filter(user_id=request.user.id,
                                               column=column_name)    #⑦
        if columns:
            return HttpResponse('2')
        else:
            ArticleColumn.objects.create(user=request.user, column=column_name)
            return HttpResponse("1")
```

在本书中已经讨论过提交表单的 CSRF 问题，这里使用语句⑥在视图函数前面添加装饰器的方式也是解决提交表单中遇到的 CSRF 问题的一种方式。

在视图函数中，用条件语句判断请求类型是 GET 还是 POST。如果是 POST，即前端提交的栏目名称，就要检验一下该名称是否已经存在，注意语句⑦的条件有两个，一个是当前用户，另一个是栏目名称。然后判断查询结果，如果数据库中没有该栏目名称，则允许创建。

根据要展示的效果，对./templates/article/column/article_column.html 中的 JavaScript 部分代码进行适当修改，修改之后的代码如下。

```
<script type="text/javascript">
    function add_column(){
        var index = layer.open({
            type:1,
            skin:"layui-layer-rim",
            area: ["400px", "200px"],
            title: "新增栏目",
            content: '<div class="text-center" style="margin-top:20px">
                        <p>请输入新的栏目名称</p>
                        <p>{{column_form.column}}</p>
                    </div>',
            btn:['确定', '取消'],
            yes: function(index, layero){
                column_name = $('#id_column').val();
                $.ajax({
                    url:'{% url "article:article_column" %}',
                    type:'POST',
                    data:{"column":column_name},
                    success:function(e){
                        if(e=="1"){
                            parent.location.reload();
                            layer.msg("good");
                        }else{
                            layer.msg("此栏目已有，请更换名称")
                        }
                    },
                });
            },
            btn2: function(index, layero){
                layer.close(index);
            }
        });
    }
</script>
```

通过这段脚本，前端可以根据来自后端的不同反馈给予不同的显示。特别提醒读者，在实际的项目中，上述代码还缺少一些东西，那就是对用户输入的内容进行判断。一般来讲，我们要限制用户输入的字符内容，比如栏目名称只允许是字母和汉字，为此需要在得到栏目名称 column_name 之后，用正则表达式来判断该名称是否合法。如果合法，就使用 Ajax 传送数据，否则提示用户重新为栏目命名。读者可以自行修改上述 JavaScript 代码，实现上述判断。

完成上述代码后，通过测试可以检查是否实现预期功能。

现在我们得到了如图 3-4 所示的结果。

图 3-4　添加栏目后的效果

添加栏目的功能已经实现，但这仅仅是栏目管理的部分功能，还有栏目的删除、编辑等功能需要添加。

3.1.2　编辑栏目

在栏目列表中，有一项“操作”列，其下面应该列出能够对本行栏目名称实施的操作。在本项目中，计划设置“删除”和“编辑”两个操作。

首先，通过修改前端./templates/article/column/article_column.html 文件，显示两种操作图标。将原来“操作”列下面对应的代码进行修改（（原来是<td>--</td>），其中“--”用如下代码代替换）。

```
<a name="edit" href="javascript:" onclick="edit_column(this, {{ column.id }})">
   <span class="glyphicon glyphicon-pencil"></span>
</a>
<a name="delete" href="javascript:" onclick="del_column(this, {{ column.id }})">
   <span class="glyphicon glyphicon-trash" style="margin-left:20px;"></span>
</a>
```

刷新 http://localhost:8000/article/article-column/页面，可以看到出现了两个图标，通常“铅笔”图标对应“编辑”功能，“垃圾桶”图标对应“删除”功能，如图 3-5 所示。

图 3-5　在操作列下有了编辑功能

为了能够实现对栏目名称的修改，要再写一个视图函数（位于./article/views.py），代码如下。

```
@login_required(login_url='/account/login')
@require_POST
@csrf_exempt
def rename_article_column(request):
    column_name = request.POST["column_name"]
    column_id = request.POST['column_id']
```

```
    try:
        line = ArticleColumn.objects.get(id=column_id)    #①
        line.column = column_name    #②
        line.save()
        return HttpResponse("1")
    except:
         return HttpResponse("0")
```

在上述代码中，多了一个装饰器@require_POST，所以一定要在文件顶部声明 from django.views.decorators.http import require_POST，使用这个装饰器的目的就是保证此视图函数只接收通过 POST 方式提交的数据。

语句①根据所要修改的栏目名称所在记录的 id，查询到该数据，并建立实例对象。语句②则实现将该属性重新赋值的功能，不要忘记 line.save()函数，否则不能保存到数据库中。

在./article/urls.py 中配置 URL，代码如下。

```
path('rename-column/', views.rename_article_column, name="rename_article_column"),
```

后端工作完成，继续修改前端模板文件。在./templates/article/column/article_column.html 文件中编写 JavaScript 代码，如下所示，实现“操作”列下的“编辑”按钮中 onclick 请求的函数功能。

```
function edit_column(the, column_id){
    var name = $(the).parents("tr").children("td").eq(1).text();
    var index = layer.open({
        type: 1,
        skin: "layui-layer-rim",
      area: ["400px", "200px"],
      title: "编辑栏目",
      content: '<div class="text-center" style="margin-top:20px">
                  <p>请编辑的栏目名称</p>
                  <p><input type="text" id="new_name" value="'+name+'"></p>
               </div>',
      btn:['确定', '取消'],
      yes: function(index, layero){
           new_name = $("#new_name").val();
            $.ajax({
            url: "{% url 'article:rename_article_column' %}",
            type: "POST",
            data: {"column_id": column_id, "column_name": new_name},
            success: function(e){
                if(e=="1"){
                    parent.location.reload();
                    layer.msg("good");
                }else{
                    layer.msg("新的名称没有保存,修改失败。")
                }
            },
          });
      },
    });
}
```

此函数与前面编写的 JavaScript 函数类似，不再解释，如有疑问，请在 Google 中查询。

下面就看看效果吧，检查 Django 服务是否在运行（python manage.py runserver），在 http://localhost:8000/article/article-column/页面中，单击代表“编辑”的铅笔图标，实现修改栏目名称的功能，如图 3-6 所示。

在弹出的对话框中修改栏目名称后，单击“确定”按钮，会有一个提示“good”闪现，意味着修改成功了，之后页面被刷新，如图 3-7 所示。

图 3-6　编辑栏目名称

图 3-7　编辑成功

修改栏目名称的功能完成了，请读者领会上述过程，按照同样的方法实现“删除”功能。

3.1.3　删除栏目

删除功能较“编辑”功能简单一些，实现方法还是一样的。

先编写视图函数，编辑./article/views.py 文件，输入如下代码。

```
@login_required(login_url='/account/login')
@require_POST
@csrf_exempt
def del_article_column(request):
    column_id = request.POST["column_id"]
    try:
        line = ArticleColumn.objects.get(id=column_id)
        line.delete()    #①
```

```
        return HttpResponse("1")
    except:
        return HttpResponse("2")
```

通过语句①删除该数据记录，然后设置 URL，在./article/urls.py 文件中增加如下代码。

```
path('del-column/', views.del_article_column, name="del_article_column"),
```

接下来在模板文件./templates/article/column/article_column.html 中编写一个名为 del_column 的 JavaScript 函数，代码如下。

```
function del_column(the, column_id){
    var name = $(the).parents("tr").children("td").eq(1).text();
    layer.open({
        type: 1,
        skin: "layui-layer-rim",
        area: ["400px", "200px"],
        title: "删除栏目",
        content: '<div class="text-center" style="margin-top:20px">
                    〈p>是否确定删除{'+name+'}栏目</p>
                </div>',
        btn:['确定', '取消'],
        yes: function(){
            $.ajax({
                url: '{% url "article:del_article_column" %}',
                type:"POST",
                data: {"column_id":column_id},
                success: function(e){
                    if(e=="1"){
                        parent.location.reload();
                        layer.msg("has been deleted.");
                    }else{
                        layer.msg("删除失败");
                    }
                },
            })
        },
    });
}
```

这是一个普通的函数，没有什么需要解释的，读者可以自行测试，看看是否能够删除已有的栏目。

如果需要将“文章栏目”的管理权限赋给超级管理员，就编辑./article/admin.py 文件，输入如下代码。

```
from django.contrib import admin
from .models import ArticleColumn

class ArticleColumnAdmin(admin.ModelAdmin):
    list_display = ('column', 'created', 'user')
    list_filter = ("column",)

admin.site.register(ArticleColumn, ArticleColumnAdmin)
```

登录管理后台 http://localhost:8000/admin/，会看到如图 3-8 所示的效果。

对栏目的管理设置差不多完成了，其实代码可以继续优化，比如前端的 JavaScript 代码中两次用到了雷同的 Ajax 方法，就可以写一个函数来完成，但是本书不讨论优化代码的问题，因为那不是我们的目标，我们的目标是了解 Django。优化代码当然重要，但是要在掌握 Django 和有关知识之后来优化代码。

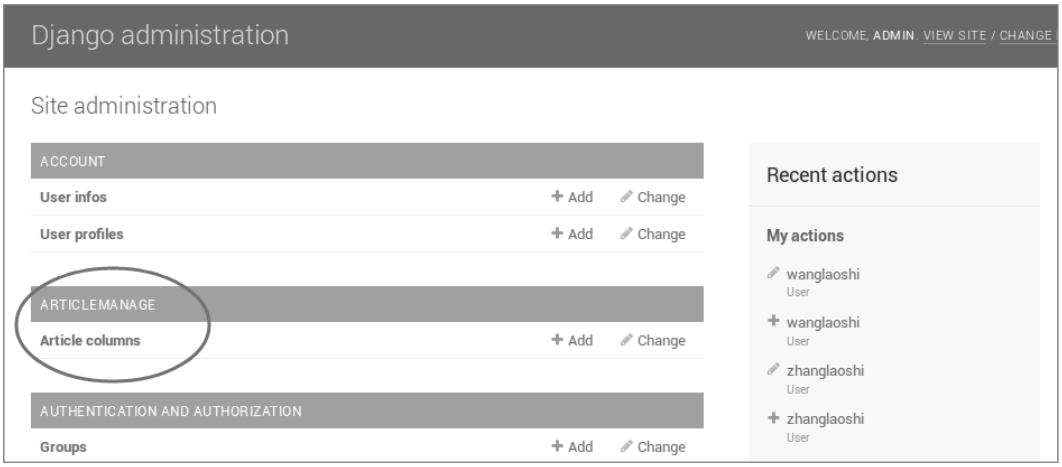

图 3-8　后台管理文章栏目

此时此刻，或许是草长莺飞，或许是秋风萧瑟，或许是雪花飘舞，或许是淫雨霏霏，不管外面如何，你都应该不为所动，掩卷细思本节的所有代码，这是一种重要的学习方法——反省（程序员的修养）。假如读者对 JavaScript 还不熟悉，就应该立刻行动，学起来。

3.1.4　知识点

1．模板语法：继承和包含

在编写前端模板时，通常会使用模板的继承和包含，这都是为了尽可能避免重复前端代码。

首先要定义一个基础模板，比如本节中用到的 base.html。在基础模板中，通常会使用很多的{% block %}，其基本格式是：

```
{% block name %}
   <!—html-->
{% endblock %}
```

每个块中可以写 HTML 代码，也可以为空。一般来说，在基础模板中定义{% block %}数量多一些，是绝对有好处的。如果将来在子模板中有相同名称（name）的块，就会将 base.html 中定义的块覆盖。

在子模板中继承的方式是使用{% extends "base.html" %}，并且放在该页面中的第一个模板标签位置，一般放在第一行最保险。在子模板中如果重写了父模板中的某个块，则按照子模板中的方式显示。

此外，在模板中还可以使用{% include "templatename.html" %}包含其他模板。

为了更灵活地使用 JavaScript 和 CSS，通常不在 base.html 里面引入相关文件。在真实的项目中，建议在 base.html 中设定{% block js %} {% endblock %}和{% block css %} {% endblock %}，然后在子模板中重写这两个块，最后调入该模板页面所需要的 JavaScript 和 CSS。

2．模型：查询

前面已经提到，Django 封装了对数据的操作，可以使用更 Python 的方式实现数据查询，而不是使用 SQL 语句。从本节开始，将陆续出现常用的数据查询方式。

```
>>> from blog.models import BlogArticles
>>> BlogArticles.objects.all()
<QuerySet [<BlogArticles: Canglaoshi is a good teacher>, <BlogArticles: Physics greate>,
<BlogArticles: Python is for you>, <BlogArticles: php is the best computer language>]>
```

BlogArticles 是一个数据模型类，可以用 Python 中常用的 dir()和 help()两个方法来研究这个类（也是一个对象）所具有的属性和方法。

```
>>> dir(BlogArticles)
['DoesNotExist', 'MultipleObjectsReturned', '__class__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_check_column_name_clashes',
'_check_field_name_clashes', '_check_fields', '_check_id_field', '_check_index_together',
'_check_local_fields', '_check_long_column_names', '_check_m2m_through_same_relationship',
'_check_managers', '_check_model', '_check_ordering', '_check_swappable',
'_check_unique_together', '_do_insert', '_do_update', '_get_FIELD_display',
'_get_next_or_previous_by_FIELD', '_get_next_or_previous_in_order', '_get_pk_val',
'_get_unique_checks', '_meta', '_perform_date_checks', '_perform_unique_checks',
'_save_parents', '_save_table', '_set_pk_val', 'author', 'author_id', 'body', 'check',
'clean', 'clean_fields', 'date_error_message', 'delete', 'from_db', 'full_clean',
'get_deferred_fields', 'get_next_by_publish', 'get_previous_by_publish', 'id',
'objects', 'pk', 'prepare_database_save', 'publish', 'refresh_from_db', 'save',
'save_base', 'serializable_value', 'title', 'unique_error_message', 'validate_unique']
```

仔细观察不难发现，上面所列出来的属性和方法包括我们在定义 BlogArticles 类时自己定义的，也包括它从父类那里继承过来的。

其中有一个 objects 属性是经常被使用的，读者可以继续用 dir(BlogArticles.objects)和 help(BlogArticles. objects)查看其详细的属性和方法。

```
Help on Manager in module django.db.models.manager object:

class Manager(BaseManagerFromQuerySet)
 |  Method resolution order:
 |      Manager
 |      BaseManagerFromQuerySet
 |      BaseManager
 |      builtins.object
#省略下面的各种方法和属性
```

这里展示的结果是一个 Manage 类（对象），每个数据模型类都有这个对象，或者说除非特别操作，每个数据模型类都有一个 objects 属性，并借着 Manager 对象中的各种属性和方法实现对数据库的基本查询操作。例如通过 BlogArticles.objects.all()查询得到的结果是一个被称为 QuerySet 对象组成的列表，列表中的每个元素就是数据库表中的一条记录，也可以理解为是 BlogArticles 类的一个实例。

```
>>> BlogArticles.objects.get(id=1)
<BlogArticles: php is the best computer language>
>>> BlogArticles.objects.get(title="Canglaoshi is a good teacher")
<BlogArticles: Canglaoshi is a good teacher>
>>> BlogArticles.objects.get(title="teacher")
Traceback (most recent call last):
  File "/usr/lib/python3.5/code.py", line 91, in runcode
    exec(code, self.locals)
  File "<console>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/django/db/models/manager.py", line 85,
in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/django/db/models/query.py", line 385, in
get
    self.model._meta.object_name
blog.models.DoesNotExist: BlogArticles matching query does not exist.
```

get()是必须要精准匹配的查询方法，如果查询内容不存在，则会报错。因为是精准匹配，所以查询到的第一个总是符合条件的记录。下面分别列出几种常用的查询方式，供读者学习参考。

- 查询所有对象

```
>>> from django.contrib.auth.models import User
>>> users = User.objects.all()
>>> users
<QuerySet [<User: admin>, <User: zhanglaoshi>, <User: wanglaoshi>, <User: zhaolaoshi>,
<User: zhoulaoshi>, <User: wulaoshi>, <User: danglaoshi>, <User: canglaoshi>, <User:
hulaoshi>, <User: yinlaoshi>, <User: lulaoshi>, <User: chenlaoshi>, <User: kuanglaoshi>,
<User: xiaodou>, <User: xiaojin>]>
```

查询结果 users 所引用的是一个序列（类列表的 QuerySet 对象，不可变），可以使用 Python 中某些对序列的操作方法来操作。

```
>>> lu = users[10]
>>> lu
<User: lulaoshi>
>>> lu.email
'lu@laoshi.com'
```

- 根据条件查询对象

在 SQL 中可以通过 where 设置查询条件，在 Django 的 QuerySet 中也有一个类似的方法 filter()。

```
>>> users_laoshi = User.objects.filter(username__endswith="laoshi")
>>> users_laoshi
```

```
<QuerySet [<User: zhanglaoshi>, <User: wanglaoshi>, <User: zhaolaoshi>, <User: zhoulaoshi>, <User: wulaoshi>, <User: danglaoshi>, <User: canglaoshi>, <User: hulaoshi>, <User: yinlaoshi>, <User: lulaoshi>, <User: chenlaoshi>, <User: kuanglaoshi>]>
```

这里查询出 username 中以“laoshi”这个字符串结尾的对象。filter()还有很多参数可以使用，读者可以参考官方文档。

除 filter()能够实现根据条件查询外，还有 exclude()和 get()，如下所示。

```
>>> user = User.objects.get(id=11)
>>> user
<User: lulaoshi>
```

get()和 filter()的区别在于，如果查询对象不存在，get()会报错，而 filter()返回空。

此外，查询操作还支持链接过滤。

```
>>> users = User.objects.filter(username__endswith="laoshi").exclude(id__gt=7)
>>> users
<QuerySet [<User: zhanglaoshi>, <User: wanglaoshi>, <User: zhaolaoshi>, <User: zhoulaoshi>, <User: wulaoshi>, <User: danglaoshi>]>
```

- 查询结果排序

对于查询返回的 QuerySet 结果，默认是按照数据模型类中定义的字段排序的，如果要重新排序，可以使用 order_by()方法。

```
>>> u = User.objects.filter(username__startswith="z").order_by("username")
>>> u
<QuerySet [<User: zhanglaoshi>, <User: zhaolaoshi>, <User: zhoulaoshi>]>
>>> u = User.objects.filter(username__startswith="z").order_by("-username")
>>> u
<QuerySet [<User: zhoulaoshi>, <User: zhaolaoshi>, <User: zhanglaoshi>]>
```

比较上面两个排序结果。

当然，order_by()的参数中可以写多个字段名称，即按照字段前后顺序分别作为排序的“主关键词”和“次关键词”。

3．文档导读

（1）*Templates*，https://docs.djangoproject.com/en/2.1/topics/templates/。

（2）*QuerySet API reference*， https://docs.djangoproject.com/en/2.1/ref/models/querysets/。

（3）*django-queryset-csv*，https://pypi.python.org/pypi/django-queryset-csv/。

3.2 发布和显示文章

对文章的管理主要是发布、删除和编辑。对于每个部分，都有一些详细的要求，本节就围绕这三个功能，开发管理文章的程序，如图 3-9 所示。

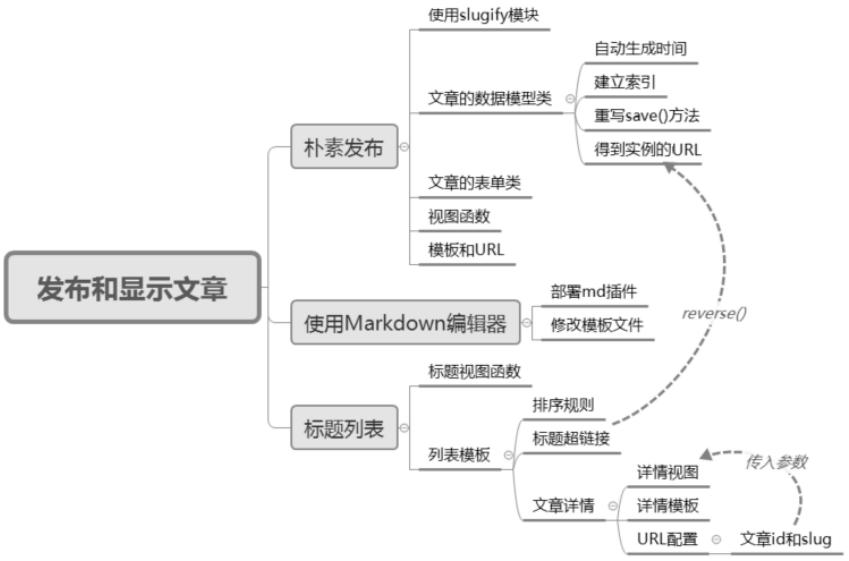

图 3-9　知识技能导图

3.2.1　简单的文章发布

首先介绍一个预备知识 slug，举例说明。

假如在数据库中有一篇标题为 *Learn Python in itdiffer.com* 的文章，要访问这篇文章，一种方法是使用这篇文章的 id，比如 http://www.itdiffer.com/article/231，231 就是这篇文章在数据库表中的 id;还有另外一种方法,是直接在 URL 中显示文章的标题(http://www.itdiffer.com/article/Learn Python in itdiffer.com)，但实际上不能这样，因为在 URL 里面，空格都是用“%20”来表示的，所以上面那个地址的最终表现形式应该是 http://www.itdiffer.com/article/Learn%20Python%20in%20itdiffer.com。这种显示不是一个很友好的方案，这时候 slug 就可以发挥作用了，它实现的结果是 http://www.itdiffer.com/article/Learn-Python-in-itdiffer.com，看起来清清爽爽了，并且这种方式对搜索引擎也是友好的。

Django 中就提供了实现上述 slug 的方法。在本项目的根目录中执行 python3 manage.py shell，进入交互模式。

```
>>> from django.utils.text import slugify
>>> slugify("Learn python in itdiffer.com")
'learn-python-in-itdiffercom'
```

表现完美。不过我们有时也会用中文写标题，不总是用英文的。那么中文怎样转换呢？

```
>>> slugify("跟老齐学")
''
```

这就不完美了，Django 内置的这个方法不能处理中文，它返回的是空。还好，有 Google 协助，很容易就搜索到了一个能够解决中文的第三方库。

```
$ pip3 install awesome-slugify
```

安装之后，还是在交互模式中，代码如下。

```
>>> from slugify import slugify
>>> slugify("跟老齐学")
'Gen-Lao-Qi-Xue'
>>> slugify("Learn python in itdiffer.com")
'Learn-python-in-itdiffer-com'
```

自动将汉字转换为拼音，这样就解决了前面遇到的汉字问题，并且英语依然适用。仔细观察，与 Django 自带的还稍微有一些区别呢！

这是一个好工具。

下面开始按套路出牌——套路就是规则，对于大的系统，按照规则来做，成本是最低的。

1．创建数据模型类

在./article/models.py 文件中建立一个数据模型，命名为 ArticlePost，作为文章的数据模型对象。

```
from django.utils import timezone    #①
from django.urls import reverse
from slugify import slugify

class ArticlePost(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name="article")
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=500)
    column = models.ForeignKey(ArticleColumn, on_delete=models.CASCADE,
related_name="article_column")
    body = models.TextField()
    created = models.DateTimeField(default=timezone.now)    #②
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ("title",)
        index_together = (('id', 'slug'),)    #③

    def __str__(self):
        return self.title

    def save(self, *args, **kargs):    #④
        self.slug = slugify(self.title)    #⑤
        super(ArticlePost, self).save(*args, **kargs)

    def get_absolute_url(self):    #⑥
        return reverse("article:article_detail", args=[self.id, self.slug])
```

代码稍多，但大多数比较熟悉，有几个“新面孔”需要解释一下。

在./mysite/settings.py 文件中，有一个 TIME_ZONE 项，读者要确认一下是否使用 TIME_ZONE = 'Asia/Shanghai'时区，语句①所引入的 timezone 将被用于语句②，default=timezong.now 即得到文章发布时的日期和时间。

语句③的作用是对数据库中这两个字段建立索引，在后面，会通过每篇文章的 id 和 slug 获取该文章对象，这样建立了索引之后，能提高读取文章对象的速度。

每个数据模型类都有一个 save 方法，语句④对此方法进行重写，其目的就是要实现语句⑤。

语句⑥是要获取某篇文章对象的 URL，这里暂时用不到，后面会用到。前面引入模块时看到的 reverse 就在这个方法中使用了。

新的数据模型建立后，就要迁移数据，从而建立数据库表。

2．创建表单类

编辑./article/forms.py 文件，增加如下代码。

```
class ArticlePostForm(forms.ModelForm):
    class Meta:
        model = ArticlePost
        fields = ("title", "body")
```

请读者在编辑上述代码时，检查一下是否已经引入了 ArticlePost 类，即在./article/forms.py 文件中写上“from .models import ArticleColumn, ArticlePost”，从而说明上述代码中的 ArticlePost 类的来源。

3．创建视图函数

编辑./article/views.py 文件，输入如下代码。

```
@login_required(login_url='/account/login')
@csrf_exempt
def article_post(request):
    if request.method=="POST":
        article_post_form = ArticlePostForm(data=request.POST)
        if article_post_form.is_valid():
            cd = article_post_form.cleaned_data
            try:
                new_article = article_post_form.save(commit=False)
                new_article.author = request.user
                new_article.column = request.user.article_column.get(
                                                id=request.POST['column_id'])
                new_article.save()
                return HttpResponse("1")
            except:
                return HttpResponse("2")
        else:
            return HttpResponse("3")
    else:
        article_post_form = ArticlePostForm()
        article_columns = request.user.article_column.all()    #⑦
        return render(request, "article/column/article_post.html",
                      {"article_post_form":article_post_form,
                        "article_columns":article_columns})
```

在视图函数中会用到 ArticlePost 类和 ArticlePostForm 类，不要忘记在本文件头部引入。

请读者将注意力集中到语句⑦，这句得到的是该登录用户已经设置的栏目对象，但为什么没有使用比较熟悉的 article_columns=ArticleColumn.objects.filter(user=request.user)呢？这么写当然可以，在此使用 request.user.article_column.all()的目的在于引起读者关注 ArticleColumn 数据

模型类中的 user=models.ForeignKey(User, related_name='article_column')，这里有一个参数 related_name，为了理解它的作用，还是启用交互模式。

```
>>> from article.models import ArticleColumn
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(id=11)
>>> user
<User: lulaoshi>
>>> user.article_column.all()
<QuerySet [<ArticleColumn: pypy>, <ArticleColumn: PHP>]>
```

user 是一个用户对象实例，在 ArticleColumn 中通过 user = models.ForeignKey(User, related_name='article_column')将 ArticleColumn 类与 User 类建立外键关系。那么，如何通过 user 实例查找到其名下 ArticleColumn 的所有实例？因为在 ForeignKey()中使用了 related_name 参数，于是就可以通过 user.article_column.all()的方式得到。

初步理解了 related_name 之后，再来看上述代码中的 request.user，在前面也用到过，这里稍微讲解一下。

当一个页面被请求时，Django 创建一个包含请求元数据的 HttpRequest 对象，然后把 HttpRequest 作为视图函数的第一个参数传入，每个视图要负责返回一个 HttpResponse 对象。这个 HttpResponse 对象有很多属性，其中包括读者熟知的 method、GET 和 POST。user 也是一个属性，在用户登录之后，request.user 即为 User 类的一个实例。

4．设置 URL

设置 URL 虽然简单，但却是关键的。编辑./article/urls.py 文件，添加如下 URL 配置语句。

```
path('article-post/', views.article_post, name="article_post"),
```

这个步骤的要诀就是简单、精准。

5．编写模板

编写模板是一个华丽的动作，但绝非华而不实。

在./templates/article/column 目录中创建一个名为 article_post.html 的文件，其代码如下。

```
{% extends "article/base.html" %}
{% load staticfiles %}
{% block title %}article column{% endblock %}
{% block content %}
<div style="margin-left:10px">
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="row" style="margin-top: 10px;">
            <div class="col-md-2 text-right"><span>标题:</span></div>
            <div class="col-md-10 text-left">{{article_post_form.title}}</div>
        </div>
        <div class="row" style="margin-top: 10px;">
            <div class="col-md-2 text-right"><span>栏目:</span></div>
            <div class="col-md-10 text-left">
              <select id="which_column">
                 {% for column in article_columns %}
```

```
                <option value="{{column.id}}">{{column.column}}</option>
                {% endfor %}
              </select>
            </div>
        </div>
        <div class="row" style="margin-top: 10px;">
          <div class="col-md-2 text-right"><span>内容:</span></div>
          <div class="col-md-10 text-left">{{article_post_form.body}}</div>
        </div>
        <div class="row">
            <input type="button" class="btn btn-primary btn-lg" value="发布"
                   onclick="publish_article()">
        </div>
    </form>
</div>
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
<script type="text/javascript">
    function publish_article(){
         var title = $("#id_title").val();
        var column_id = $("#which_column").val();
        var body = $("#id_body").val();
        $.ajax({
          url: "{% url 'article:article_post' %}",
          type: "POST",
          data: {"title":title, "body":body, "column_id":column_id},
          success: function(e){
              if(e=="1"){
                  layer.msg("successful");
              }else if(e=="2"){
                  layer.msg("sorry.") ;
              }else{
                  layer.msg("项目名称必须写，不能空。");
              }
          },
        });
    }
</script>
{% endblock %}
```

这个模板中的代码并不陌生，建议读者一行一行地阅读，并且在旁边写注释，从而真正理解。

前端模板还没有结束，还要修改./templates/article/leftslider.html 文件，目的是设置一个发布文章的入口。

```
<p><a href="{% url 'article:article_post'%}">发布文章</a></p>
```

将上面这行代码放在 leftslider.html 文件的合适位置。

6．测试本功能

确保 Django 服务已运行，在浏览器的地址栏中输入 http://localhost:8000/article/article-post/，然后在打开的页面中输入要发布的内容，如图 3-10 所示。

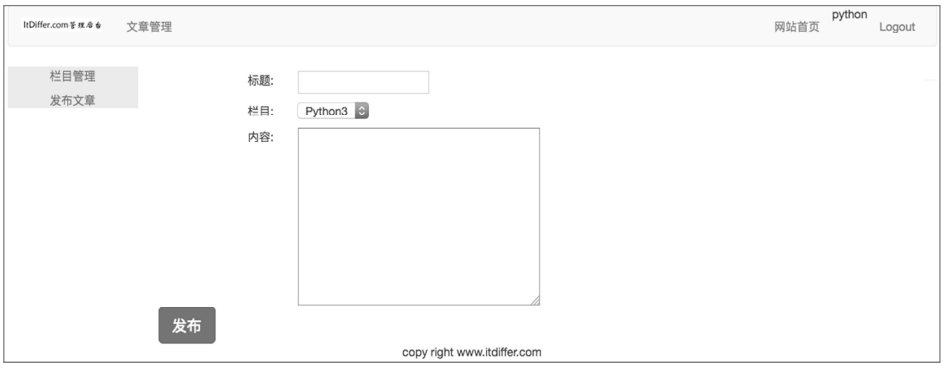

图 3-10　发布文章界面

界面的确很朴素，连按钮的位置都没有好好安排，而且在文本框中只能输入文本。不过，重点是看功能是否实现，单击“发布”按钮看一下效果。

弹出一个对话框，上面有“successful”，表示成功了；如果报错，要仔细排查，对照上述代码，必要时“请”出 Google，或者根据本书最开始列出的其他方式求助。

经过上面 6 个步骤，就完成了简单的文章发布功能的开发。

接下来就可以美化一下了。

3.2.2　使用 Markdown

Markdown 是最好的编辑器之一，笔者推荐并推崇它。使用 Markdown 编辑器，可以让编写的内容“豪华”起来。

Markdown 是一种轻量级的标记语言，由 John Gruber 于 2004 年创立。虽然是一种“标记语言”（HTML 也是标记语言），但是用起来并不难，也不需要用户痛苦地记忆，因为它语法简单、标记符数量少，更重要的是可以完全可视化操作，就像使用某种可视化的文档编辑工具一样。

在本项目中，我们就使用这样一个 Markdown 编辑器，其来源是 https://github.com/qiwsir/editor.md。在此对这个 Markdown 插件的开发者 pandao 表示感谢。

下面就创建我们的豪华 Markdown 编辑器。

1．部署相关文件

从 GitHub 上把刚才推荐的插件下载并解压缩，然后在本项目的 static 目录中建立一个子目录 editor，从已经解压缩的 editor.md-master 目录中，复制一些目录到刚刚建立的./static/editor 目录中，所复制的项目如图 3-11 所示。

图 3-11　markdown 的目录和文件

2．修改模板

这个插件主要是基于前端的，所以只需要修改./templates/article/column/article_post.html 文件。首先，在文件中引入两个 CSS 文件。

```
<link rel="stylesheet" href="{% static 'editor/css/style.css' %}">
<link rel="stylesheet" href="{% static 'editor/css/editormd.css' %}">
```

其中，editormd.css 文件是所下载的插件自带的，另外一个 style.css 文件是笔者根据所下载的插件中的例子改写的，代码如下。

```
#layout {
    text-align: left;
}

#layout > header, .btns {
    padding: 15px 0;
    width: 90%;
    margin: 0 auto;
}

#layout > header > h1 {
    font-size: 20px;
    margin-bottom: 10px;
}
```

在./templates/article/column/article_post.html 文件中找到如下代码。

```
<div class="col-md-10 text-left">{{article_post_form.body}}</div>
```

用下面的代码替换上面找到的代码。

```
<div id="editormd" class="col-md-10 text-left">
    <textarea style="display:none;" id="id_body"></textarea>
</div>
```

最后在模板文件尾部引入 JavaScript 文件和相应的脚本代码。

```
<script type="text/javascript" src='{% static "editor/editormd.min.js" %}'></script>
<script type="text/javascript">
    $(function() {
        var editor = editormd("editormd", {
            width   : "100%",
            height  : 640,
            syncScrolling : "single",
            path    : "{% static 'editor/lib/' %}"
        });
    });
</script>
```

可以查看效果了。确保 Django 服务在运行，并且用户处于登录状态，比如还是使用前面已经出现的账号登录。在浏览器的地址栏中输入 http://localhost:8000/article/article-post/（如果用户没有登录，系统会提示登录），就可以看到发布文章的界面，如图 3-12 所示。

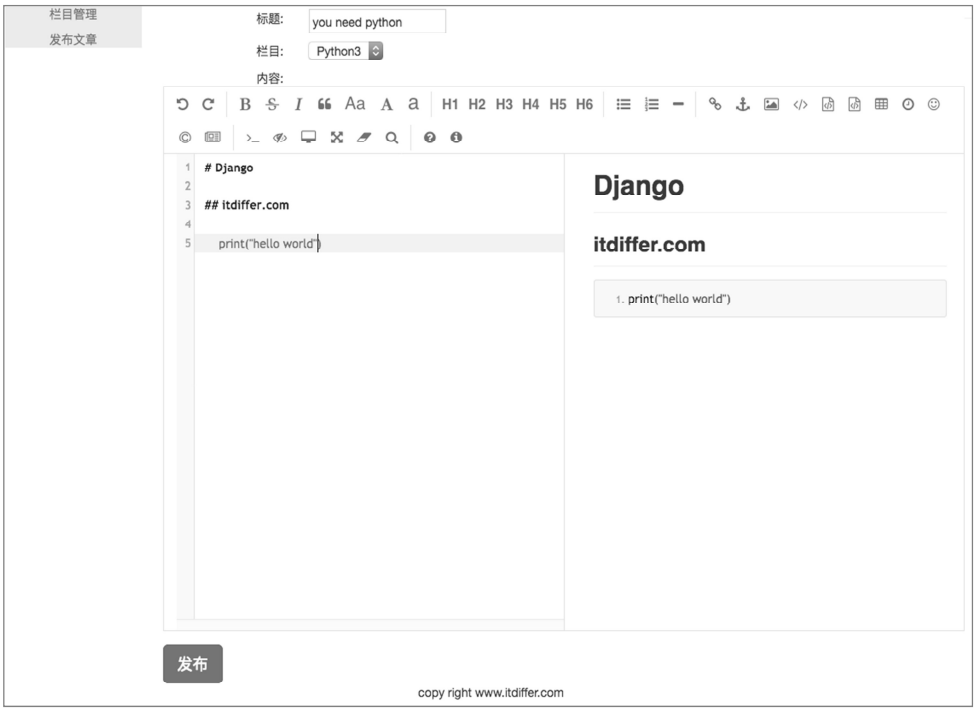

图 3-12　Markdown 编辑器效果

酷！只能用一个字来表达激动的心情了。

呈现在显示器上的是一个功能丰富的图文混排的编辑器，当然这个编辑器是我们所使用的 Editor 插件中最简单的。根据 Markdown 的语法输入内容，在左边输入内容后，在右边可以看到显示效果。

通过这种方式发布的文章，就不再是简单的文本了，而是可以包含图片、文字、音频、视频等多媒体文档。特别推荐读者使用 Markdown 的标记符号，能够让你编辑文档更快捷。

通过单击“发布”按钮，就能够把文章发布，但是还不能浏览，接下来就要显示文章及其标题列表。

3.2.3　文章标题列表

对于显示文章列表的功能，读者应该有一点基础了，还记得第 1 章中简单的博客吗？那里我们已经涉及一些类似的功能了。不过，这里要做得更漂亮些。

1．简单的标题列表

俗话说“不能一口吃个胖子”，所以还是从简单的开始。编辑./article/views.py 文件，编写一个简单的视图函数。

```
@login_required(login_url='/account/login')
def article_list(request):
    articles = ArticlePost.objects.filter(author=request.user)    #①
    return render(request, "article/column/article_list.html", {"articles":articles})
```

用语句①筛选出用户的所有文章对象，并将该对象渲染到模板。

创建模板文件./templates/article/column/article_list.html，输入如下代码。

```
{% extends "article/base.html" %}

{% load staticfiles %}

{% block title %}articles list{% endblock %}

{% block content %}
<div>
    <table class="table table-hover">
     <tr>
          <td>序号</td>
          <td>标题</td>
          <td>栏目</td>
          <td>操作</td>
     </tr>
     {% for article in articles %}
     <tr id={{ article.id }}>
          <td>{{ forloop.counter }}</td>
          <td>{{ article.title }}</td>
          <td>{{ article.column }}</td>
          <td>--</td>
     </tr>
     {% endfor %}
    </table>
</div>
{% endblock %}
```

上述模板代码与文章栏目名称列表代码类似，实现了简单的标题列表。最后一步就是设置 URL，编辑./article/urls.py 文件，增加下面的代码。

```
path('article-list/', views.article_list, name="article_list"),
```

因为新增加了文件，所以要重启 Django 服务，在浏览器的地址栏中输入 http://localhost:8000/article/article-list/，效果如图 3-13 所示。

图 3-13　博客文章列表页

这个效果非常好！

如果读者在调试时没有看到预期的效果，那么一定要认真阅读界面给出的提示信息，它是排查错误的重要依据。

多发布几篇文章，观察一下在图 3-13 所示的页面中，这些文章是按照什么顺序排列的。

请读者翻阅前面写的 ArticlePost 类，其中有下面的代码。

```
class Meta:
    ordering = ("title",)
    index_together = (('id', 'slug'),)
```

如果以前没有体会到 ordering = ("title",)的作用，现在应该体会到了，仔细观察文章标题列表的结果，是不是按照标题名称的顺序排列的？的确是。

如果要修改为按照文章编辑/发布的时间倒序排列，就应该修改 ordering 的值，代码如下。

```
ordering = ("-updated",)
```

保存之后，再次刷新页面，就看到所期望的结果了。

为了访问方便，左侧的功能栏当然要增加显示文章标题的入口，编辑./templates/article/leftslider.html 文件，在读者已经熟悉的位置增加下面的代码。

```
<p><a href="{% url 'article:article_list' %}">文章列表</a></p>
```

再次刷新浏览器页面，效果如图 3-14 所示。

图 3-14　修改左侧导航栏

很漂亮！越来越趋近完美的目标了。

再接再厉！

2．查看文章

文章标题列出来了，接下来就是为其设置超链接，单击标题后就能查看文章内容。在第 1 章中我们通过文章的 id 查看详细内容，这里要换一种方式查看。

请读者再次阅读 ArticlePost 类的代码，里面有如下内容。

```
def get_absolute_url(self):
    return reverse("article:article_detail", args=[self.id, self.slug])
```

这段代码中使用了 reverse()函数，先进行望文生义，reverse 有“颠倒、倒转”之意。因为 Python 中的函数都是依据其中文意义命名的，所以这个函数必然有此功能，只不过它是“倒转”什么呢？这就需要查看官方文档来解惑了，以下解释参考了官方文档（https://docs.djangoproject.com/en/2.0/ref/urlresolvers/）。

reverse()函数的调用方式是：

```
reverse(viewname, urlconf=None, args=None, kwargs=None, current_app=None)
```

参数 viewname 就是在每个应用的 urls.py 中设置 URL 时 name 的值。比如，前面刚刚在./article/urls.py 中增加的 URL（path('article-list/', views.article_list, name="article_list")）配置中的 name="article_list"。

还是以上述 URL 为例，如果要向该视图函数发出请求，可以使用如 localhost:8000/article/article-list/的形式，其中 localhost:8000 是域名和端口部分，而/article/article-list/就是路径部分，严格说是请求资源的 URI（Uniform Resource Identifier，统一资源定位符）。

在模板文件中，遇到了需要使用链接地址时，可以使用{% url viewsname %}的形式，比如：

```
<a href='{% url "article:article_list" %}'>文章列表</a>
```

其实，上述代码也可以写成如下形式。

```
<a href='/article/article-list/'>文章列表</a>
```

为了避免硬编码问题，所以采用前述方式，只要规定的名称不变化，路径设置的修改不会影响这里的超链接对象的访问。

这是在模板中，如果在视图函数中呢？比如：

```
HttpResponseRedirect('/article/article-list/')
```

如果这样写，也是“硬编码”的风格，要避免。代替它的就是使用 reverse()函数。

```
HttpResponseRedirect(reverse('article:article_list'))
```

通过 reverse('article:article_list')实现了'/article/article-list/'，即从 name 到 path，而在 URL 配置中是从 path 到 name，因此 reverse()起到了“逆向、颠倒”的作用。

reverse()参数中除 viewsname 外，还可以传入其他数值，比如前面的 reverse("article:article_detail", args=[self.id, self.slug])。

如果读者打算深入了解这个函数的执行过程，还可以看一下该函数的源码，网址是 https://docs.djangoproject.com/en/2.1/ref/urlresolvers/。

在 ArticlePost 类中的 get_absolute_url()方法，就是要得到相应文章的路径，因此可以将其用于模板文件./templates/article/column/article_list.html 的<a>标签中。

```
<td><a href="{{ article.get_absolute_url }}">{{ article.title }}</a></td>
```

这样修改模板文件，实现的效果是让每个标题都实现超链接，链接对象是该文章的详细内容。

刷新页面，如果 Django 服务已经运行，那么就会看到如图 3-15 所示的报错信息，这没有什么意外。

图 3-15　文章列表页报错

“学海无涯苦作舟，书山有路勤为径。”学习的过程都是痛苦的，只有学习的结果才能够带来快乐。在学习编程的过程中，会有很多“坑”，当你用最大的努力将这些“坑”都填平了，才能体会到其中的快乐。

面对刚刚发现的“坑”，最好的办法就是心平气和地认真阅读提示信息。“NoReverseMatch at /article/article-list/”，这意味着 URL 配置存在问题。另外，从图中信息可知，self.id 和 self.slug 已经被读取到，但是没有对应的 URL 来接收这两个值。所以，要增加 article_detail 的 URL 配置，用来显示文章的详情。在./article/urls.py 中增加如下 URL 配置代码。

```
re_path('article-detail/(?P<id>\d+)/(?P<slug>[-\w]+)/$',
        views.article_detail,  name="article_detail"),
```

这种写法以前没有见到过，其中包含了两个参数<id>和<slug>，并且用正则表达式约定了相应参数的形式，与之对应的视图函数 article_detail()就应该包含这两个参数。另外，没有使用熟悉的 path()，而是用了 re_path()，所以需要在文件顶部导入这个函数，即增加：

```
from  django.urls import path, re_path
```

关于 path()和 re_path()的比较，请阅读 https://docs.djangoproject.com/en/2.0/ref/urls/中的相关内容。通过比较不难发现，re_path()的用法相当于 Django 1.x 中的 url()。

编辑./article/views. py 文件，增加如下函数。

```
from django.shortcuts import get_object_or_404

@login_required(login_url='/account/login')
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    return render(request, "article/column/article_detail.html", {"article":article})
```

同时还要再次编写./templates/article/column/article_detail.html 模板文件，为了快速地查看效果，先写一个简单的——这里暂时不要纠结内容是否按照输入格式显示问题。

```
{% extends "article/base.html" %}

{% block title %}articles list{% endblock %}

{% block content %}
<div>
```

```
<h1>{{ article.title }}</h1>
<p>{{ user.username }}</p>
<div>
    {{ article.body }}
</div>
</div>
{% endblock %}
```

刷新文章列表页面，可以看到每个文章标题都已经做好了超链接，如图 3-16 所示。

图 3-16　修正后的文章列表

单击某个标题，可以查看该标题所对应的文章，如图 3-17 所示。

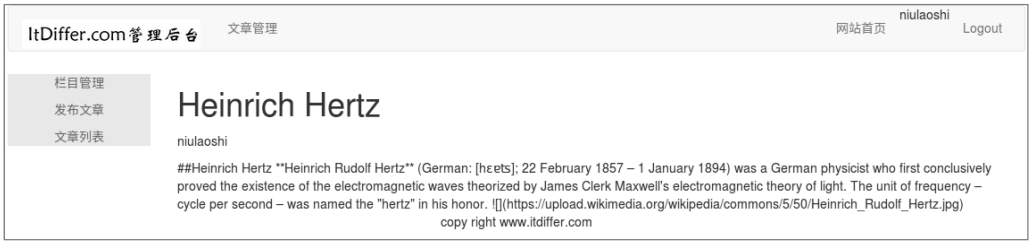

图 3-17　文章详情

虽然文章内容的显示方式没有进行美化，但不要在乎其外表，还是要看本质，已经实现了基本功能。此外，还要特别观察一下 URL 的特点，与我们所设计的模式正好符合。

3．按排版格式显示

尽管已经能够看到所发布的内容了，但是在显示方式上并没有按照发布文章时所设定的格式显示。所以，还需要继续编辑显示文章内容的模板文件./templates/article/column/article_detail.html，主要是将 Markdown 标记符转换为 HTML 标记符并显示在网页上，重新编辑之后的代码如下。

```
{% extends "article/base.html" %}

{% load staticfiles %}

{% block title %}articles list{% endblock %}

{% block content %}
```

```
<div>
    <header>
        <h1>{{ article.title }}</h1>
        <p>{{ user.username }}</p>
    </header>

    <link rel="stylesheet" href="{% static 'editor/css/editormd.preview.css' %}" />
    <div id='editormd-view'>
        <textarea id="append-test" style="display:none;">
{{ article.body }}
          </textarea>
    </div>

</div>
<script src='{% static "js/jquery.js" %}'></script>
<script src='{% static "editor/lib/marked.min.js" %}'></script>
<script src='{% static "editor/lib/prettify.min.js" %}'></script>
<script src='{% static "editor/lib/raphael.min.js" %}'></script>
<script src='{% static "editor/lib/underscore.min.js" %}'></script>
<script src='{% static "editor/lib/sequence-diagram.min.js" %}'></script>
<script src='{% static "editor/lib/flowchart.min.js" %}''></script>
<script src='{% static "editor/lib/jquery.flowchart.min.js" %}'></script>
<script src='{% static "editor/editormd.js" %}'></script>

<script type="text/javascript">
$(function(){
       editormd.markdownToHTML("editormd-view", {
       htmlDecode      : "style,script,iframe",  // you can filter tags decode
       emoji           : true,
       taskList        : true,
       tex             : true,  // 默认不解析
       flowChart       : true,  // 默认不解析
       sequenceDiagram : true,  // 默认不解析
    });
});
</script>
{% endblock %}
```

对于学习编程，笔者一直建议不要复制代码，哪怕是最简单的代码，也要自己敲，因为在敲代码的时候，会遇到很多意想不到的问题，甚至会遇到恼人的拼写错误。对于上述代码，笔者在测试过程中就遇到了一个“坑”，用了好长时间才解决，那就是写{{ article.body }}代码时不要像编写 HTML 代码时那样缩进，前面不能有空格，因为它带过来的是 Markdown 标记，不能按照 HTML 的编码格式写。读者可以试试，看缩进几个空格是什么效果。

上述代码中引入了大量的 JavaScript 文件，都是我们所使用的 Markdown 插件所提供的。

再次刷新页面，所看到的就应该是一个按照排版要求显示的页面了，如图 3-18 所示。

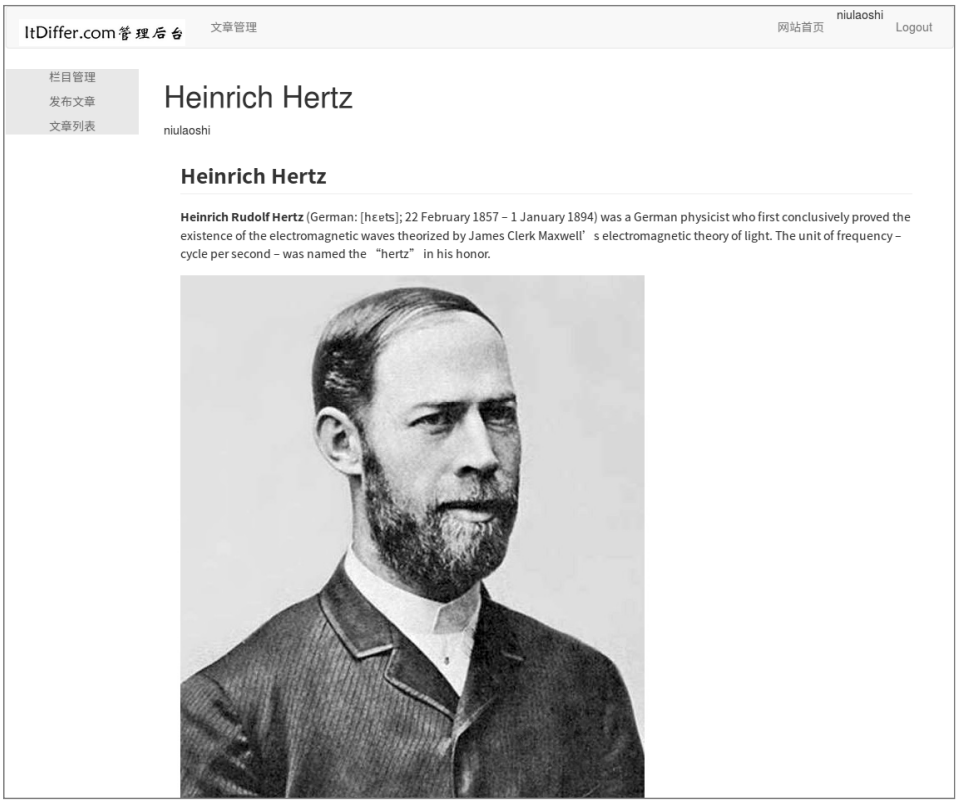

图 3-18　按照排版格式显示

然而，“发布文章”的功能还要优化，目前成功发布一篇文章之后，仅仅是提示成功，页面并没有跳转。

4．发布文章后页面的跳转

文章发布之后，页面应该跳转到“文章列表”，这样就可以通过列表中的文章标题查看该文章的内容，当然也可以跳转到文章内容页，下面演示的是跳转到文章标题列表页面，读者可以在学习了下面的内容之后，自己实现跳转到该文章内容页的功能。

需要修改的仅仅是./templates/article/column/article_post.html 中的 JavaScript 部分代码。

```
#其他代码
<script type="text/javascript">
    function publish_article(){
        #其他代码从略
        success: function(e){
            if(e=="1"){
                layer.msg("successful");
                location.href = "{% url 'article:article_list' %}";    #①
            }
            #省略后续代码
</script>
```

在视图文件中，文章被保存后向前端反馈，前端则通过 JavaScript 脚本捕获反馈信息并进

行判断，语句①就是根据文章已经被正确保存的反馈信息做出的页面跳转操作。

读者完成上述编码之后，自行测试，看看能否实现预想的功能。

查看已经做好的“栏目管理”、“发布文章”和“文章列表”三个功能，然后查看文章详情，应该很有成就感了。一个不错的具有内容发布和管理功能的系统已经有了雏形——关键这是一个刚刚开始学习 Django 的你做的，所以，完全可以自豪一下了。

自豪之后，还要回到眼前的代码中，因为还有功能没有完成，“文章列表”中的“操作”一项下面还是空的，要补充对文章的删除和编辑功能——这就是程序员的工作，总是“路在前方”。

3.2.4 知识点

1．关于 slug

slug 的目的在于为每条记录生成一个 URL，并且让这个 URL 更易读。比如一篇文章的 URL，有的是 http://www.itdiffer.com/course/38，这里使用文章在数据库中的 id（id=38），这种方式虽然能够显示对应内容，但是不容易阅读；如果用类似 http://www.itdiffer.com/course/learn-django-with-laoqi 的样式，那么这个 URL 的可读性就很好了。当然，不能仅仅如此，因为文章标题有可能重复，最好是 http://www.itdiffer.com/course/38/learn-django-with-laoqi。这里的“learn-django-with-laoqi”就是要保存在数据库中的 slug。为此，在数据模型的字段属性中有专门的一个属性 models.SlugField()，这个属性的源码读者可以阅读 https://docs.djangoproject.com/en/1.10/_modules/django/db/models/fields/#SlugField，它其实也是 CharField 的子类。Django 这么安排，对发布文章的永久链接是非常友好的（不要忘记 Django 的起源）。

但是，上述的所有阐述是针对英文标题的，如果在中文环境中，则可以使用本节中的方法转换为汉语拼音。

2．模型：“一对多”

“一对多”这种说法翻译自“one-to-many”，不管是翻译还是原文，都容易造成一种误解，误以为有方向性，即误以为“一对多”和“多对一”是有区别的，而真实的“一对多”没有方向性。比如用户和文章的关系，一个用户可以发表多篇文章和多篇文章可以对应一个用户，是一样的关系。

在数据库中，为了建立两个表之间的“一对多”的关联，要使用外键。所谓外键（Foreign Key），是用于建立两个数据库表之间的链接的一个或多个字段。比如有一个数据库表 A 和用户数据库表 User 建立了外键关系，在表 A 中有一个名为 user_id 的字段，它就是表 A 的外键。当表 A 的数据被保存时，会自动在 user_id 记录中保存 User 表中用户的 id，用这种方式创建了数据库表 A 和用户表 User 之间的链接，也就明确了表 A 中的每条记录是属于哪一个用户的。

Django 的数据模型类中的某个字段的属性被设置为某个表的外键后，在数据库表中，以“字段名”+“_id”的方式命名一个记录，并保存关联数据库表中记录的 id 值（或者理解为关联数据模型类实例的 id）。图 3-19 所示就是本节所创建的数据模型类 ArticlePost 对应的数据库表结构。

Columns (8)

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	id	integer	1	null	1
1	title	varchar(200)	1	null	0
2	slug	varchar(500)	1	null	0
3	body	text	1	null	0
4	updated	datetime	1	null	0
5	author_id	integer	1	null	0
6	column_id	integer	1	null	0
7	created	datetime	1	null	0

Name　Type　Not Null　Default

图 3-19　数据库表结构

外键有几个常用的参数，能够为数据访问带来很多便利，下面仅列出两个。

- limit_choices_to，专门用于为字段设置选项，例如 staff_member = models.ForeignKey(User, limit_choices_to={'is_staff': True})（此示例引自官方文档）。
- related_name，用于关联对象的反向查询，比如在 ArticlePost 类中的 author = models.ForeignKey(User, related_name="article")，可以通过 User 实例的 article 属性得到该用户的所有 ArticlePost 实例（某个用户的所有文章）。这种反向查询在本书后续项目中会经常用到，读者在实践中可以逐渐理解。如果在这里定义了 related_name，那么使用“_set”进行的查询就不能使用了，这一点请读者注意。

3．文档导读

（1）*Many-to-one relationships*，https://docs.djangoproject.com/en/2.1/topics/db/examples/many_to_one/。

（2）*ForeignKey*，https://docs.djangoproject.com/en/2.1/ref/models/fields/#foreignkey。

（3）*Making queries*，https://docs.djangoproject.com/en/2.1/topics/db/queries/。

3.3　删除和修改文章

网站上对文章的操作通常简称为“增删改查”：“增”即发布，“删”即删除，“改”即修改，“查”即查看。继上节内容，本节学习“删改”之法，如图 3-20 所示。

图 3-20　知识技能导图

3.3.1　删除

如同“栏目管理”中的功能，对文章也有删除和修改的操作。还是采用 bootstrap 默认的两

个图标，先在./templates/article/column/article_list.html 文件中的相应<td>内将该图标显示出来，代码如下。

```
<span class="glyphicon glyphicon-pencil"></span>
<span class="glyphicon glyphicon-trash" style="margin-left:20px;"></span>
```

这样“操作”项目下也显示了编辑和删除图标，如图 3-21 所示。

图 3-21　增加操作项目的列表

实现“删除”操作相对简单一些，这样容易使读者建立信心。

这次换一个开发顺序，不再先写视图函数了（当然数据模型和表单类都不用写了），而是“从前向后”先写“删除”的前端功能。因此，还要继续编辑上述模板文件，在“删除”功能的代码外围加上<a>标签。

```
<a nane="delete" href="javascript:" onclick="del_article(this, {{ article.id }})"> <span
class="glyphicon glyphicon-trash" style="margin-left:20px;"></span></a>
```

显然需要写一个提交要删除的文章 id 的 JavaScript 脚本。

```
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
<script type="text/javascript">
    function del_article(the, article_id){
        var article_name = $(the).parents("tr").children("td").eq(1).text();
        layer.open({
            type: 1,
            skin: "layui-layer-rim",
            area: ["400px", "200px"],
            title: "删除文章",
            content: '<div class="text-center" style="margin-top:20px">
                        <p>是否确定删除《'+article_name+'》</p>
                    </div>',
            btn:['确定', '取消'],
            yes: function(){
                $.ajax({
                    url: '{% url "article:del_article" %}',
                    type:"POST",
```

```
                    data: {"article_id":article_id},
                    success: function(e){
                        if(e=="1"){
                            parent.location.reload();
                            layer.msg("has been deleted.");
                        }else{
                            layer.msg("删除失败");
                        }

                    },
                })
            },
        });
    }
</script>
```

这里依然使用那个熟悉的弹出提示框插件，的确很好用。

前端设置完成后，就要编写删除文章的视图函数了，编辑./article/views.py 文件，增加del_article()函数。

```
@login_required(login_url='/account/login')
@require_POST
@csrf_exempt
def del_article(request):
    article_id = request.POST['article_id']
    try:
        article = ArticlePost.objects.get(id=article_id)
        article.delete()
        return HttpResponse("1")
    except:
        return HttpResponse("2")
```

别忘了，还要在./article/urls.py 文件中对删除操作进行 URL 设置。

```
path('del-article/', views.del_article, name="del_article"),
```

现在就可以测试一下删除文章的功能了，应该不会出错，万一调试不过，请耐心检查。

3.3.2　修改

在编写修改文章的功能之前，先梳理清楚实现这个功能的流程。

（1）单击修改文章的图标，将该文章的 id 传给相应的视图函数。

（2）视图函数从数据库中读取该文章 id 的相关内容。

（3）将文章相关内容呈现在页面上，并且处于编辑状态，要求依然使用 Markdown 编辑器。

（4）用户编辑之后，单击“提交”按钮，再次将页面中编辑的内容保存在上述 id 所对应的数据库记录中。

从上述流程中可以看出，我们必须编写一个视图函数（依然在./article/views.py 文件中），这个视图函数能够接收文章的 id，而且支持 GET 和 POST 两种请求方式。先编写满足 GET 请

求的部分，代码如下。

```
@login_required(login_url='/account/login')
@csrf_exempt
def redit_article(request, article_id):    #①
    if request.method == "GET":
        article_columns = request.user.article_column.all()
        article = ArticlePost.objects.get(id=article_id)
        this_article_form = ArticlePostForm(initial={"title":article.title})
        this_article_column = article.column
        return render(request, "article/column/redit_article.html",
                      {"article":article,
                       "article_columns":article_columns,
                      "this_article_column":this_article_column,
                      "this_article_form":this_article_form})
```

重点关注语句①的参数，除熟知的 request 外，又增加了 article_id，这就是用来接收文章 id 的。正是通过 article_id，才能获得该文章对象。

接着在./templates/article/column/目录中创建名为 redit_article.html 的文件，这就是上述视图函数所指定的前端模板文件，其代码如下。

```
{% extends "article/base.html" %}

{% load staticfiles %}

{% block title %}article column{% endblock %}

{% block content %}
<link rel="stylesheet" href="{% static 'editor/css/style.css' %}">
<link rel="stylesheet" href="{% static 'editor/css/editormd.css' %}">
<div class="container">
<div class="col-md-10">
<div style="margin-left:10px">
    <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="row" style="margin-top: 10px;">
            <div class="col-md-2 text-right"><span>标题:</span></div>
            <div class="col-md-10 text-left">{{this_article_form.title}}</div>
        </div>
        <div class="row" style="margin-top: 10px;">
            <div class="col-md-2 text-right"><span>栏目:</span></div>
            <div class="col-md-10 text-left">
              <select id="which_column">
                  {% for column in article_columns %}
                       {% if column == this_article_column.column %}    #②
                         <option value="{{column.id}}" selected="selected">
                            {{column.column}}
                         </option>    #③
                       {% else %}
                          <option value="{{column.id}}">{{column.column}}</option>
                       {% endif %}
                  {% endfor %}
```

```
            </select>
          </div>
       </div>
       <div  class="row" style="margin-top: 10px;">
        <div class="col-md-2 text-right"><span>内容:</span></div>
        <div id="editormd" class="col-md-10 text-left">
             <!--{{article_post_form.body}}-->
             <textarea style="display:none;" id="id_body">
{{article.body}}    #④
             </textarea>
        </div>
       </div>
       <div class="row">
          <input type="button" class="btn btn-primary btn-lg" value="发布"
                 onclick="redit_article()">
       </div>
   </form>

</div>
</div>
</div>
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript" src='{% static "editor/editormd.min.js" %}'></script>
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
<script type="text/javascript">
   $(function() {
       var editor = editormd("editormd", {
          width   : "100%",
          height  : 640,
          //syncScrolling : "single",
          path    : "{% static 'editor/lib/' %}",
       });
   });
</script>

{% endblock %}
```

因为此文章在发布时已经选定栏目了，所以在循环体内部要通过语句②判断当前文章原有的栏目名称，并将下拉菜单中的该名称确定为选定状态，即语句③。

语句④得到文章的内容，但是要注意，前面还是不要留有空格，原因见前文叙述。

下面配置 URL，在./article/urls.py 文件中增加如下内容。

```
path('redit-article/<int:article_id>/', views.redit_article, name="redit_article"),
```

检查 Django 服务是否已经启动，在浏览器的地址栏中输入 http://localhost:8000/article/redit-article/5/，最后的数字就是某篇文章的 id，读者可以根据自己的项目输入其他数值，但不能省略，也不能输入不存在的 id 值，效果如图 3-22 所示。

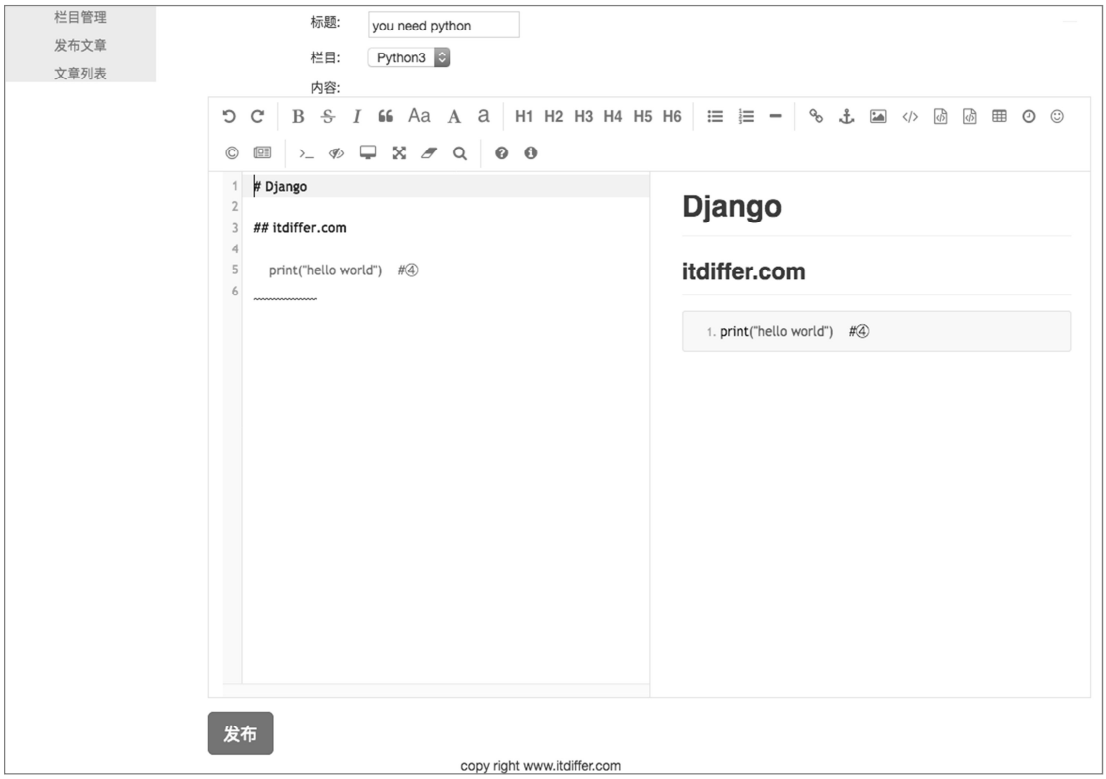

图 3-22　编辑已经发布的文章

现在就可以在本页面中编辑文章了。

至此，完成了编辑文章的一部分功能。但是，总不能让用户记住自己每篇文章的 id，所以要再做一个单击“铅笔”图标就能够跳转到上述界面的功能，需要再次对./templates/article/column/article_list.html 文件进行编辑。

前面已经给“删除”图标加上了<a>标签，实现了删除功能。现在再给“编辑”图标加上<a>标签，实现编辑功能，代码如下。

```
<a name="edit" href="{% url 'article:redit_article' article.id %}">
   <span class="glyphicon glyphicon-pencil"></span>
</a>
```

{% url 'article:redit_article' article.id %}这个写法较以往的 URL 写法稍有不同，后面多了一个参数 article.id，它对应 URL 中的 article_id。

完成之后，刷新 http://localhost:8000/article/article-list/页面，单击“编辑”图标，检查是否跳转到上述编辑文章的界面。

必须是，否则就不能继续下去了；如果不是或者报错，请根据报错信息进行检查。

就差最后一步了。在文章编辑界面单击“发布”按钮时，将数据提交给视图函数，还是用熟悉的 Ajax 方法。其实，在模板文件中已经有了伏笔 onclick="redit_article()"，在./templates/article/column/redit_article.html 的尾部编写 redit_article()函数，代码如下。

```
<script type="text/javascript">
```

```
    function redit_article(){
        var title = $("#id_title").val();
       var column_id = $("#which_column").val();
       var body = $("#id_body").val();
       $.ajax({
         url: "{% url 'article:redit_article' article.id %}",
         type: "POST",
         data: {"title":title, "body":body, "column_id":column_id},
         success: function(e){
             if(e=="1"){
                 layer.msg("successful");
                  location.href = "{% url 'article:article_list' %}";
             }else{
                 layer.msg("sorry.");
             }
         },
       });
    }
</script>
```

在 POST 对象地址中，使用了{% url 'article:redit_article' article.id %}，这种形式的 URL 不仅可以用在 HTML 代码中，还可以用在 JavaScript 代码中。

前面./article/views.py 中的视图函数 redit_article()只编写了响应 GET 请求的部分，现在编写响应 POST 请求的部分，对此视图函数增加如下代码。

```
@login_required(login_url='/account/login')
@csrf_exempt
def redit_article(request, article_id):
    #省略部分代码
    else:
        redit_article = ArticlePost.objects.get(id=article_id)
        try:
            redit_article.column = request.user.article_column.get(
                                                    id=request.POST['column_id'])
            redit_article.title = request.POST['title']
            redit_article.body = request.POST['body']
            redit_article.save()
            return HttpResponse("1")
        except:
             return HttpResponse("2")
```

这样就将本小节开头所说的修改文章的功能都实现了。测试一下吧！从文章列表界面单击“编辑”图标，进入文章编辑界面，对文章进行修改，再单击“发布”按钮，看看是否实现了修改的目的。祝你一路单击顺利，没有 Bug。

3.3.3　设置分页功能

在文章标题列表中，应该再增加一个分页功能，因为在发布的文章多了之后，在同一个页面中显示出所有的标题是要耗费较长时间的，所以分页是必需的。

Django 也认为分页是常见的操作，所以提供了内置的分页方法，可以在视图函数./article/views.py 的顶部引入分页功能用到的三个类。

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
```

下面重写 article_list()视图函数，代码如下。

```
@login_required(login_url='/account/login')
def article_list(request):
    articles_list = ArticlePost.objects.filter(author=request.user)
    paginator = Paginator(articles_list, 2)    #①
    page = request.GET.get('page')    #②
    try:
        current_page = paginator.page(page)    #③
        articles = current_page.object_list    #④
    except PageNotAnInteger:    #⑤
        current_page = paginator.page(1)
        articles = current_page.object_list
    except EmptyPage:    #⑥
        current_page = paginator.page(paginator.num_pages)    #⑦
        articles = current_page.object_list
    return render(request, "article/column/article_list.html",
                  {"articles":articles, "page": current_page})
```

语句①根据所查询到的文章对象 articles_list 创建分页的实例对象，并且规定每页最多 2 个。Paginator 类的初始化的完整参数列表是 Paginator(object_list, per_page, orphans=0, allow_empty_first_page=True)，详细代码可以参考 https://docs.djangoproject.com/en/2.1/topics/pagination/#django.core.paginator.Paginator。

语句②获得当前浏览器 GET 请求的参数 page 的值，也就是当前浏览器所请求的页码数值。在后面的前端模板中，我们会看到，通过浏览器发出的 URL 请求中都包含 page 参数，即 URL 的样式诸如 http://localhost:8000/article/article-list/?page=1，通过语句②得到 URL 中的参数 page 的值 1，即当前所请求的页码是 1（第 1 页），并赋值给 page 变量。

语句③中的 page()是 Paginator 对象的一个方法，其作用在于得到指定页面内容，其参数必须是大于或等于 1 的整数。

语句④object_list 是 Page 对象的属性，能够得到该页所有的对象列表。类似的属性还有 Page.number（返回页码）等。

语句⑤和语句⑥捕获两个异常，一个是请求的页码数值不是整数（PageNotAnInteger），另一个是请求的页码数值为空或者在 URL 参数中没有 page（EmptyPage）。

语句⑦paginator.num_pages 返回的是页数，num_pages 是 Paginator 对象的一个属性。

重写了视图函数之后，还要再写一个专门显示分页的模板文件，创建./templates/paginator.html 文件，并输入如下代码。

```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
    <a href="?page={{ page.previous_page_number }}">Previous</a>
```

```
    {% endif %}
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}
    </span>
    {% if page.has_next %}
    <a href="?page={{ page.next_page_number }}">Next</a>
    {% endif %}
  </span>
</div>
```

在上述代码中，使用了 Page 对象的几个属性。

- has_previous：判断是否有上一页。
- previous_page_number：返回上一页的页码。
- number：返回当前页的页码。
- paginator.num_pages：关联 Paginator 对象（page.paginator），并得到其总页码数（paginator.num_pages）。
- next_page_number：返回下一页的页码。

完成分页模板的代码编写之后，将它引入到 article_list.html 文件中，编辑./templates/article/column/article_list.html 文件，并在表格代码<table></table>的后面加入如下内容。

```
{% include "paginator.html" %}
```

检查 Django 服务是否已经运行，为了看到效果，请读者先在文章发布页面中多发布几篇文章，然后访问 http://localhost:8000/article/article-list/，就会看到一个简明且全面的分页功能，如图 3-23 所示。

图 3-23　有分页效果的列表页

对内容维护所需要的“增删改查”功能都已经实现了。读者已经越来越体会到编程的乐趣了吧！的确，程序员是快乐的，因为总是能够通过自己的双手创造喜欢的作品。

3.3.4　知识点

1．Markdown

在本项目中，编辑文章所用的编辑器是 Markdown，没有用别的编辑器。Markdown 编辑器是目前最好的编辑器之一。

感谢这个编辑器的创始人 John Gruber。

Markdown 是一种可以使用普通文本编辑器编写的标记语言，通过简单的标记语法，可以使普通文本内容具有一定的格式。Markdown 具有一系列衍生版本，用于扩展 Markdown 的功能（如表格、脚注、内嵌 HTML 等），这些功能原始的 Markdown 尚不具备。而衍生版本能让 Markdown 转换成更多的格式，例如 LaTeX、Docbook。Markdown 增强版中比较有名的有 Markdown Extra、MultiMarkdown、Maruku 等。这些衍生版本要么基于工具，如 Pandoc；要么基于网站，如 GitHub 和 Wikipedia。它们在语法上基本兼容，但在一些渲染效果上有所不同。

关于 Markdown 的语法，可以查看 http://wowubuntu.com/markdown/，或者访问官方网站 http://daringfireball.net/projects/markdown/。

现在，如果不用 Makdown，就落伍了。

2．模型：插入和更新

SQL 语句中的 INSERT 和 UPDATE 可以实现数据库表中记录的插入和更新。在 ORM 中，要实现这种操作，可以使用完全类似 Python 中创建实例或者属性赋值的方式进行。

（1）插入数据

```
>>> from django.contrib.auth.models import User
>>> new_user = User(username="xiaodou", password="1qaz0okm")
>>> new_user.save()
```

上述代码等效于 SQL 语句中的 INSERT，最后要使用 new_user 实例的 save()方法才能完成数据的入库操作。除这种方法外，还可以使用如下方式。

```
>>> User.objects.create(username="xiaojin", password="1qaz0okm")
<User: xiaojin>
```

create()函数直接让数据入库。

（2）修改数据

```
>>> lu.email = "lulaoshi@itdiffer.com"
>>> lu.save()
>>> lu.email
'lulaoshi@itdiffer.com'
```

注意，save()函数是必需的。

从上述操作可以看出，插入和更新操作不返回 QuerySet 结果，这与查询操作是有显著差别的。

3．模型：删除

ORM 中删除记录的方法非常简单。

```
>>> one = User.objects.get(username="zhoulaoshi")    #①
>>> one
<User: zhoulaoshi>
>>> one.delete()    #②
(1, {'auth.User_user_permissions': 0, 'blog.BlogArticles': 0, 'course.Course_student':
0, 'auth.User': 1, 'course.Lesson': 0, 'account.UserProfile': 0, 'account.UserInfo': 0,
'auth.User_groups':  0,  'image.Image':  0,  'admin.LogEntry':  0,  'ArticleManage.
ArticlePost_users_like': 0})
>>> one = User.objects.filter(username="zhoulaoshi")    #③
```

```
>>> one
<QuerySet []>
>>> one = User.objects.get(username="zhoulaoshi")    #④
Traceback (most recent call last):
  File "/usr/lib/python3.5/code.py", line 91, in runcode
    exec(code, self.locals)
  File "<console>", line 1, in <module>
  File "/usr/local/lib/python3.5/dist-packages/django/db/models/manager.py", line 85,
in manager_method
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/django/db/models/query.py", line 385, in get
    self.model._meta.object_name
django.contrib.auth.models.DoesNotExist: User matching query does not exist.
```

语句①创建了 User 实例之后，语句②直接执行该实例对象的 delete()方法即可将实例本身删除，即删除数据库表中的相应记录。删除后返回该记录（实例）的值。

语句③使用 filter()方法查询，返回空；语句④使用 get()方法，因为已经不存在该记录了，所以报错。

4．文档导读

（1）*jQuery Pagination plugin*，https://github.com/esimakin/twbs-pagination。

（2）*django-markdown*，https://pypi.python.org/pypi/django-markdown。

（3）*QuerySet API reference*，https://docs.djangoproject.com/en/2.1/ref/models/querysets/。

3.4　文章展示

请回忆已经完成的功能，前文已经实现了用户注册、登录、发布文章，以及超级管理员的后台管理。下面要做的就是向访问网站的用户展示作者及其文章，如图 3-24 所示。

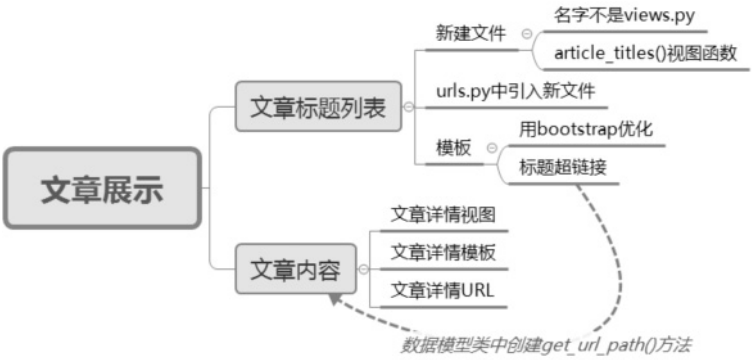

图 3-24　知识技能导图

3.4.1　新写文章标题列表

前面已经编写过文章标题列表，这里要“新”写，肯定与前面的不同。不过，基本的逻辑还是一样的，不同之处在于展示效果。

首先要编写的是视图函数，这次要新建一个编写视图函数的文件./article/list_views.py，这样做的目的主要是让读者理解视图函数的文件不一定都是 views.py，然后创建一个展示文章标题的视图函数 article_titles()，其代码如下。

```
from django.shortcuts import render
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
from .models import ArticleColumn, ArticlePost

def article_titles(request):
    articles_title = ArticlePost.objects.all()
    paginator = Paginator(articles_title, 2)
    page = request.GET.get('page')
    try:
        current_page = paginator.page(page)
        articles = current_page.object_list
    except PageNotAnInteger:
        current_page = paginator.page(1)
        articles = current_page.object_list
    except EmptyPage:
        current_page = paginator.page(paginator.num_pages)
        articles = current_page.object_list

    return render(request, "article/list/article_titles.html",
                  {"articles":articles, "page": current_page})
```

这个视图函数的代码都是之前学习过的，请读者自行为每行代码写注释，从而理解其含义，如果遇到问题，请复习前面的内容。

下面配置本应用的 URL，就是创建./article/urls.py 文件，并输入如下代码。

```
from . import views, list_views    #①

urlpatterns = [
   path('list-article-titles/', list_views.article_titles, name="article_titles"),
]
```

因为视图函数 article_titles()在 list_views.py 文件中，所以要通过语句①引入 list_views。

接下来是编写前端模板文件。在./templates/article 目录中创建 list 目录，再创建 article_titles.html 模板文件，即./templates/article/list/article_titles.html，其代码如下。

```
{% extends "base.html" %}

{% block title %} articles {% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
   <h1>阅读，丰富头脑，善化行为</h1>
</div>
<div class="container">
   {% for article in articles %}
   <div class="list-group">
       <a href="#" class="list-group-item active">
```

```
        <h4 class="list-group-item-heading">{{article.title}}</h4>
        <p class="list-group-item-text">作者：{{article.author.username}}</p>    #①
        <p class="list-group-item-text">概要：{{article.body|slice:'70'| linebreaks}}
        </p>  #②
       </a>
    </div>
    {% endfor %}
{% include "paginator.html" %}    #③
</div>
{% endblock %}
```

语句①中的变量用于得到该文章作者的用户名。在./article/models.py 中我们定义了 ArticlePost 数据模型类，其中属性 author = models.ForeignKey(User, related_name="article")，所以就能够以{{article.author.username}}方式得到用户名。

语句②（<p>标签所包含的）中的{{ article.body }}用于显示该文章对象的所有文章内容，但是后面用管道符“|”对显示的内容做了限制（有一种常见的说法叫作“过滤器”），“slice:'70'”的含义是将前面的变量所导入的内容“切下”前 70 个字符。“slice”是根据字符数量来截取部分内容的，与之类似的还有一个 truncatewords，读者不妨将语句②中的变量更换为{{article.body|truncatewords:'70'| linebreaks}}，看看有什么效果。silce 是“切下”一定数量的字符，从内容的开头计算，只要是一个字符就算一个，空格也算一个字符；而 truncatewords 则是截取一定数量的 words（单词），在英语中 words 之间用空格分隔，所以它会根据空格进行截取，但是遇到中文就麻烦了，中文不用空格分开各个词语，除非分段的地方。所以，如果读者按照笔者所说的更换了，就会看到对中文基本没有截取（读者的文章估计没有 70 个段落）。为了测试出上面说的效果，读者不仅要发布几篇英文文章，也要发布几篇中文文章。

语句②中还有一个过滤器 linebreaks，其意义从名字上可以推断一下。读者可以把它删除，看一下效果。linebreaks 的作用是允许原文中的换行 HTML 标记符继续产生效用。

语句③是很熟悉的内容，将前面已经写好的分页模板引入。

检查 Django 服务是否重新启动，在浏览器的地址栏中输入 http://localhost:8000/article/list-article-titles/地址，不管用户是否登录，都能看到如图 3-25 所示的效果。

图 3-25　文章列表页

列表完成后，下面要查看文章内容。

其实前面编写过查看文章内容的功能，为了叙述完整，同时也是跟读者一起复习，下面快速地重复一遍。

在./article/list_views.py 文件中增加如下视图函数（不要忘记在文件顶部引入 get_object_or_ 404）。

```
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    return render(request, "article/list/article_content.html", {"article":article})
```

上述代码中的 article_detail()函数与以往不同，它不检查用户是否处于登录状态。

还是熟悉的步骤，在./article/urls.py 文件中增加如下 URL 配置代码。

```
path('article-content/<int:id>/<slug:slug>/', list_views.article_detail,
     name="article_content"),
```

在 urls.py 文件中，请读者将上面的 URL 和前面已经有的一个 URL 配置（re_path(r'^article-detail/(?P<id>\d+)/(?P<slug>[-\w]+)/$', views.article_detail, name="article_detail"),）进行对比。两个 URL 配置因为使用了不同的函数（paht()和 re_path()），所以在写法上就有了很大的区别。另外，虽然都有 article_detail()函数，但因为在不同模块（views/list_views）中，所以两个函数互不影响。注意，name 的值要区分开，在同一个 URL 配置文件中不要出现相同的 name。

在./templates/article/list 中创建模板文件 article_content.html，代码如下。

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}articles detail{% endblock %}

{% block content %}
<div class="container">
    <header>
        <h1>{{ article.title }}</h1>
        <p>{{ article.author.username }}</p>
    </header>

    <link rel="stylesheet" href="{% static 'editor/css/editormd.preview.css' %}" />
    <div id='editormd-view'>
        <textarea id="append-test" style="display:none;">
{{ article.body }}
         </textarea>
    </div>
</div>
<script src='{% static "js/jquery.js" %}'></script>
<script src='{% static "editor/lib/marked.min.js" %}'></script>
<script src='{% static "editor/lib/prettify.min.js" %}'></script>
<script src='{% static "editor/lib/raphael.min.js" %}'></script>
<script src='{% static "editor/lib/underscore.min.js" %}'></script>
<script src='{% static "editor/lib/sequence-diagram.min.js" %}'></script>
```

```
<script src='{% static "editor/lib/flowchart.min.js" %}''></script>
<script src='{% static "editor/lib/jquery.flowchart.min.js" %}'></script>
<script src='{% static "editor/editormd.js" %}'></script>

<script type="text/javascript">
$(function(){
    editormd.markdownToHTML("editormd-view", {
       htmlDecode      : "style,script,iframe",  // you can filter tags decode
       emoji           : true,
       taskList        : true,
       tex             : true,  // 默认不解析
       flowChart       : true,  // 默认不解析
       sequenceDiagram : true,  // 默认不解析
    });
});
</script>
{% endblock %}
```

再编辑./templates/article/list/article_titles.html 模板文件，在文章标题外面的<a>标签中增加超链接对象，代码如下。

```
<a href="{{article.get_absolute_url}}" class="list-group-item active">
```

一切都与以往一样，没有什么特别的。如果现在处于登录状态，请退出，否则测试不到下述现象。对于浏览文章，我们约定任何人都可以。

访问 http://localhost:8000/article/list-article-titles/页面，把鼠标移到文章标题上，先不要单击，看浏览器底端，一般会有对象地址，显示的超链接地址是不是预想的那个呢？应该不是。根据前面的 URL 配置，文章详细内容的地址应该是符合类似“http://localhost:8000/article/article-content/2/learn-python”的样式，以符合 URL 配置中“article-content/<int:id>/<slug:slug>/”的要求，而不是 http://localhost:8000/article/article-detail/4/how-to-learn-program/的样式，这种样式是用户发布文章后查看详情的链接地址。并且，如果此时点击页面中的文章标题，会跳转到登录窗口。

导致上述现象的原因是在./templates/article/list/article_titles.html 文件中使用了{{article.get_absolute_url}}，而深藏在./article/models.py 中的 ArticlePost 类的 get_absolute_url()方法中有 reverse("article:article_detail", args=[self.id, self.slug]),“article:article_detail”导致了上述跳转对象。

3.4.2　重新编写“查看文章”功能

直接套用时遇到了问题，不能再使用{{article.get_absolute_url}}。

如何修改呢？

既然问题的根源是 ArticlePost 类中的 get_absolute_url()方法，那么就可以弃之不用，再写一个方法。将如下方法追加到./article/models.py 的 ArticlePost 类中。

```
def get_url_path(self):
    return reverse("article:article_content", args=[self.id, self.slug])    #①
```

语句①的写法和 return reverse("article:article_detail", args=[self.id, self.slug])的效果是一样的，只不过语句①的 URL 对象发生了变化。

将./article/list/article_titles.html 中的超链接修改为：

```
<a href="{{article.get_url_path}}" class="list-group-item active">
```

再次访问文章列表页 http://localhost:8000/article/list-article-titles/，单击文章标题，这时候应该能够看到文章详细内容了。如果报错，请耐心地检查错误原因，特别要非常认真地查看显示的各种错误信息。面对错误，自己不要太自信，“我没有错呀，按照书上说的一步一步操作的呀”，这是初学者最容易有的心态。而现实是摆在面前的，错了就是错了，计算机不会在这里撒谎的。所以要认真检查，比如拼写是否正确等。

貌似到这里就已经能够实现所有功能了。但是，这种方法其实存在一个隐患。在文章的 URL 中，我们使用的是数据库自增的 id，这就是隐患，如果因为某种原因，自增 id 发生了变化，比如合并数据库表（可能是低概率，就怕万一），这时候再按照原来的 URL 访问，就不能找到那篇文章了。就一般情况而言，网上发布的文章，应该具有一个永久的 URL，所以，应该给每篇文章增加一个单独的固定 id（或者别的标识），通过这个固定的 id（标识）访问该文章。本书暂不按照此想法修改实例，读者可以自行研究如何实现上述要求。

3.4.3 知识点

1．模板：过滤器

Django 模板中的过滤器，常常根据需要在模板中显示变量形式时使用。其基本样式是{{ value | filter }}，其中 filter 是过滤器。Django 模板提供了多种针对变量数值的过滤器。

- capfirst，将变量的第一个字符转换为大写，如果第一个字符不是字母则过滤器失效。例如{{ value|capfirst }}，如果 value 是“django”，则输出“Django”。
- cut，移除变量中的字符。例如{{ value|cut:" " }}，如果 value 是“learn django”，则输出“learndjango”。
- date，根据指定的格式输出时间。例如{{ value|date:"D d M Y" }}，如果 value 是 datetime 对象的实例（比如常用的 datetime.datetime.now()），则输出类似于“Web 09 Jan 2008”样式的字符串。官方文档中列出了各种格式字符的说明，读者可参阅。
- join，使用字符连接列表元素，其效果类似 Python 中的 str.join(list)。例如{{ value|join:" // " }}，如果 value 是['a', 'b', 'c']，则会输出“a // b // c”。

以上内容是根据官方文档的说明整理出来的，在官方文档中还有很多，请读者在用到的时候去查阅。

这些都是内置的过滤器，在某些时候还需要自定义。本书后面的项目中也会演示如何自定义一个过滤器。

2．表单：表单类

在模板中，由<form></form>组成表单，里面是一个一个的<input>，这个表单通过<form>规定的方式提交到 Django 的视图部分，通常使用 POST 或者 GET 方式。本书中的项目都使用了 POST。视图得到提交来的表单数据之后，在处理这些表单数据时，可以使用一个与数据模型（Model）类类似的类，即 Form（表单）类。当然，不用也可以，但用了更简洁。

表单类也是一个普通的 Python 类，这个类里面定义了一些变量（类的属性），这些变量分别对应着 HTML 中的<input>。我们知道，在<input>中有不同的类型，从而规定是文本输入框还是按钮等。在表单类中，也是通过类似在数据模型类中声明字段类型的方式，规定了变量的类型。

```
from django import forms
class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

通过前面的编程实践，我们已经认识到，数据模型类中的字段及其类型、表单类中的变量及其类型、模板中的<input>及其类型，都是一一对应的，否则数据无法正确保存。因此，如果没有特别需要，在表单类中可以直接使用内部类 class Meta 声明与数据模型类中对应的变量。例如：

```
class UserInfoForm(forms.ModelForm):
    class Meta:
        model = UserInfo
        fields = ("school", "company", "profession", "address", "aboutme", "photo")
```

在内部类 Meta 中，以 model = UserInfo 说明表单中各个字段的来源，然后用 fields 变量说明在本表单类中用到的字段，如果选择所有字段，则可以使用 fields = "__all__"，还可以使用 exclude = ["school"]的方式排除某些字段。

3．文档导读

（1）*Working with forms*，https://docs.djangoproject.com/en/2.1/topics/forms/。

（2）*Built-in template tags and filters*，https://docs.djangoproject.com/en/2.1/ref/templates/builtins/。

（3）*django.urls utility functions*，https://docs.djangoproject.com/en/2.1/ref/urlresolvers/。

第4章

拓展网站功能

网站如同一个复杂的世界，当程序员辛辛苦苦地实现了最基本的“增删改查”之后，就会有人以各种名义提出更多的需求，看看如今的网站，哪个不是让人眼花缭乱，甚至某些周边的功能已经被认为是网站标配了。

诗人顾城说：“草在结它的种子，风在摇它的叶子。我们站着，不说话，就十分美好。”程序员多么希望“我们坐着/站着，不敲代码，就十分美好”，怎奈那美好只属于诗人，敲代码、向人们展现程序的魅力，是程序员的职业。

4.1 梳理已有功能

对于一般的网站，我们都希望输入主域名就能访问网站的主页。在生产环境中，可以通过服务器中的 Nginx 或者 Apache 来配置网站的首页。但是，我们目前的项目使用的是开发环境，可以暂时不处理这个问题，如果非要处理，也不是不行，并且难度不大，相信读者能够自己搞定。本节要做的是梳理本项目已有的功能，让它们访问更方便，如图 4-1 所示。

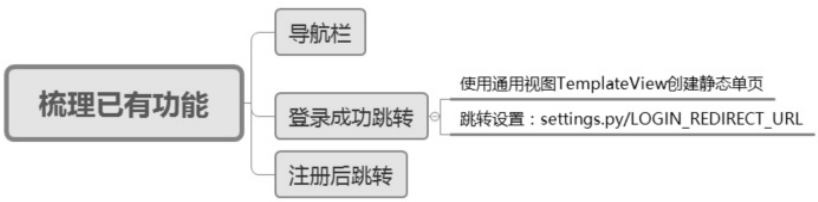

图 4-1　知识技能导图

4.1.1 修改导航栏

在前面的开发中，一直使用的那个导航栏完全是刚开始为了占位而定的，现在是修改它的时候了。编辑./templates/header.html 文件，修改的部分代码如下。

```
#未修改代码从略
<ul class="nav navbar-nav" role="navigation">
```

```
<li><a href="{% url 'blog:blog_title' %}">BLOG</a></li>
<li><a href="{% url 'article:article_titles' %}">文章</a></li>   #新增
</ul>
#未修改代码从略
```

修改之后，访问 http://localhost:8000/blog/，会看到新的导航栏如图 4-2 所示，单击“文章”选项后，进入本项目所有文章的列表页面。

图 4-2　增加了“文档”的导航栏

导航栏上有登录入口，但是缺少注册入口，请读者看一下前面关于注册的内容，是不是一直在使用某个 URL 访问呢？下面就为导航栏增加一个注册入口，代码如下。

```
<li><a href="{% url 'account:user_register' %}">Sign up</a></li>
```

将上述代码放置在判断“LOGIN”或“Logout”状态的语句后面，当刷新 http://localhost:8000/blog/页面后，会看到如图 4-3 所示的效果。

图 4-3　增加了“Sign up”的导航栏

上述操作其实是在为以往做的功能设置入口。在开发中，有不少功能只要我们稍加改造，跳出程序员的圈子，换个角度想想，就能给用户一种耳目一新的感觉。

4.1.2　修改登录和注册后的跳转

在本项目中，已经完成了用户登录和注册功能，但是，完成这两个功能之后，页面跳转到哪里？前面没有认真对待，现在要补上。

1．网站首页

单击导航栏右侧的“LOGIN”进入登录页面，但是登录之后，页面跳转到第 1 章中的内容，这不是本项目需要的，应该跳转到本网站认定的页面。所以，先做一个页面，作为本网站首页。

笔者设想的这个首页是一个有图片的静态网页，本来可以使用前面学习过的知识完成，但是本着学习的目的，换一种方式，借用后面要学习的“通用视图”来制作。

在./mysite 的 urls.py 文件中，引入通用视图 TemplateView，代码如下。

```
from django.views.generic import TemplateView
```

然后在本文件的 URL 映射表中增加如下配置代码。

```
path('home/', TemplateView.as_view(template_name="home.html"), name='home'),
```

请读者暂且按照上面的步骤来做，至于其中的缘由，后面会揭晓的。

接下来在./templates/目录中创建 home.html 文件，并编写如下代码。

```
{% extends "base.html" %}
{% block title %}Home page{% endblock %}
```

```
{% block content %}
<div class="text-center">
    <h1>WELCOM YOU</h1>
    <h3>Life is short. You need Python</h3>
    <h2>Django makes it easier to build better Web apps more quickly and with less
code.</h2>
    {% load staticfiles %}
    <img src="{% static 'images/book.jpg' %}">
</div>
{% endblock %}
```

在保证 Django 服务运行的情况下，可以通过 http://localhost:8000/home/查看刚才所做页面的效果，如图 4-4 所示。

图 4-4　网站首页

请再看一下上面的流程，我们没有写视图函数，只使用了 Django 自带的 TemplateView，就实现了首页的制作。是不是很奇妙？这种奇妙在使用 Django 开发过程中，还会不断地体验到。

通常网站的导航栏中都要有“首页”字样，点击它就跳转到图 4-4 所示的页面。为实现此功能，请修改./templates/header.html 文件，增加如下导航内容的代码。

```
<li><a href="{% url 'home' %}">HOME</a></li>
```

根据图 4-5 所示的效果，读者应该知晓上述代码的安放位置了。

图 4-5　修订之后的导航栏

还记得之前./mysite/settings.py 文件中的 LOGIN_REDIRECT_URL = '/blog/'吗？这个设置的目的就是要实现登录之后的跳转目标页，现在修改它就能达成前述功能要求，代码如下。

```
LOGIN_REDIRECT_URL = '/home/'
```

如此简单。

如果已经登录，请退出后再登录，测试一下是否跳转到了刚刚设置的首页。

2．注册后跳转页面

在原来的功能中，注册后没有跳转页面，只是返回一个提示“successfully”。当然，注册失败也是类似的。

现在就要修改一下，如果用户注册成功，自动跳转到登录页面，等待用户登录。

修改./account/views.py 文件中的 register()方法，首先引入一个我们已经使用过的函数 reverse()，代码如下。

```
from django.urls import reverse    #./account/views.py 头部引入
```

然后将 register()函数中的 return HttpResponse("successfully")用下面的代码替换。

```
return HttpResponseRedirect(reverse("account:user_login"))
```

这样，当用户注册成功之后，就跳转到登录页面了——请读者自行测试修改之后的效果。

其实，对 register()函数的其他返回也可以进行修改，这个功能就留给聪明的读者尝试了。

再来审视一下本项目，是不是已经在各种功能上感觉顺畅了很多？如果不是，建议读者仿照上述流程进行修改。笔者所做的暂时就这么多了，虽然可能距离产品经理的要求还很远。

4.1.3　知识点

1．关于 Git

在软件开发中，版本控制是必需的，因为作为一名程序员，在编程时会遇到各种不可预知的问题，需要利用版本控制来维护源码、记录改动、回滚历史、设置不同分支等，并且使不同级别的人对代码有不同的控制权限。

对于版本控制，有的人会认为等于复制。如果不严格地说，也将就着算吧，但这绝对不是程序员的方法，程序员的方法是要使用某种版本控制系统实现对版本的管理。

关于版本控制系统的历史，读者可以通过查阅资料自己去了解。这里只说当前，而且重点推荐笔者喜欢的版本控制系统——Git。

Git 的发明人是曾经在 1991 年创建了 Linux 的 Linus。想了解和学习 Git，可以登录官网（https://git-scm.com/）学习。当然，也有很多其他的相关资料，读者可自选。

仅有 Git 还不是程序员的终极目标。2007 年，又出现了一个 GitHub，它是一个通过 Git 进行版本控制的软件源代码托管服务网站，是当今世界上大型的代码存放网站和开源社区之一。在 GitHub 之后，亦有一些模仿者出现。

笔者在这里建议读者到 github.com 上注册，学习如何使用，并且把随同本书建立的项目代码存放到 GitHub。这是有百利而无一害的事情，一定要做。

2．文档导读

（1）*Settings*，https://docs.djangoproject.com/en/2.1/ref/settings/。

（2）*Django settings*，https://docs.djangoproject.com/en/2.1/topics/settings/。

（3）*Git*，https://en.wikipedia.org/wiki/Git。

4.2 查看作者全部文章

现在已经能够非常方便地查看所有文章了，但读者不要忘记，我们现在做的不是第 1 章中那种单独给某个人使用的博客系统，而是一个多用户的系统了，即每个注册用户都可以发布文章。那么，如果读者想查看某个作者的全部文章，该如何做？这是一个非常合理的功能需求，本节就来实现这个功能，如图 4-6 所示。

既然要查阅某个作者的文章，那么就需要在请求时提供该作者的用户名，这种查找方式我们已经有经验了，读者如果忘记了过去的经验，请查阅本书前面的相关内容。

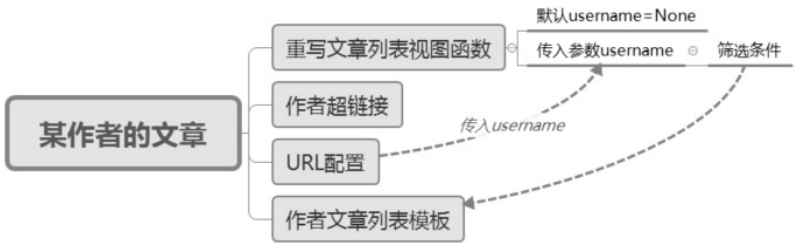

图 4-6　知识技能导图

4.2.1 查看某作者的文章列表

首先要修改./article/list_views.py 文件中的 article_titles()视图函数，增加如下代码。

```
from django.contrib.auth.models import User
def article_titles(request, username=None):    #①
    #以下代码插入原代码之前
    if username:
        user = User.objects.get(username=username)    #②
        articles_title = ArticlePost.objects.filter(author=user)    #③
    else:
        articles_title = ArticlePost.objects.all()    #这是原代码中已有的一行
#后接原来代码的其他部分内容，省略
```

语句①的参数列表用了 username=None，这种方式兼顾两种可能，一种是直接请求所有文章标题，另外一种是请求某个用户（作者）的文章标题。

语句②用于得到某个用户（作者）对象，并且用在语句③中，因为在 ArticlePost 类中，有 author = models.ForeignKey(User, on_delete=models.CASCADE, related_name="article")。

在模板文件中对作者名字增加超链接，实现单击作者名字查看该作者全部文章标题的功能。编辑./templates/article/list/article_titles.html 文件，for 循环块内的代码修改如下。

```
<div class="list-group">
    <a href="{{article.get_url_path}}" class="list-group-item active">
        <h4 class="list-group-item-heading">{{article.title}}</h4>
    </a>
    <p class="list-group-item-text">作者：<a href="{% url 'article:author_articles'
article.author.username %}">{{article.author.username}}</a></p>
    <p class="list-group-item-text">概要：{{article.body|slice:'70'| linebreaks}}</p>
</div>
```

这里依然使用了熟悉的{% url 'article:author_articles' article.author.username %}方式，并通过 article.author.username 传递 username 的值。

接着在./article/urls.py 文件中增加 URL 配置代码。

```
path('list-article-titles/<username>/', list_views.article_titles,
     name="author_articles"),
```

上述步骤完成后，检查 Django 服务是否已经运行。在服务运行的情况下访问项目，单击导航栏中的“文章”选项，可以查看所有文章标题。在每篇文章标题下面，作者的名称已经是超链接了，单击它可以查看该作者的全部文章标题，如图 4-7 所示。

请读者认真观察浏览器地址栏中 URL 的特点。

至此，还没有结束。当用户查看某篇文章内容时，里面也有作者名称，还要做同样的链接。所以，要编辑./templates/article/list/article_detail.html 文件，找到显示用户名的 HTML 标记符，用跟上面一样的处理方法，增加<a href="{% url 'article:author_articles' article.author.username %}">超链接。

图 4-7　某作者的文章列表

不要忘记保存。刷新页面后测试新功能，这样就实现了本节开头所提出的功能需求。

其实，这里所呈现的作者文章标题列表页可以继续美化，比如显示该作者的有关信息，我们曾经在用户个人信息设置中让用户填写了不少内容呢！要实现这个功能，就需要对视图函数进行重写，再次编辑./article/list_views.py 文件中的 article_titles()函数。为了避免读者在反复修改时出现差错，下面列出全部代码。

```
def article_titles(request, username=None):
    if username:
        user = User.objects.get(username=username)
        articles_title = ArticlePost.objects.filter(author=user)
        try:
            userinfo = user.userinfo    #④
        except:
            userinfo = None
    else:
        articles_title = ArticlePost.objects.all()
```

```
    paginator = Paginator(articles_title, 2)
    page = request.GET.get('page')
    try:
        current_page = paginator.page(page)
        articles = current_page.object_list
    except PageNotAnInteger:
        current_page = paginator.page(1)
        articles = current_page.object_list
    except EmptyPage:
        current_page = paginator.page(paginator.num_pages)
        articles = current_page.object_list

    if username:
        return render(request, "article/list/author_articles.html",
                      {"articles":articles, "page":current_page,
                       "userinfo":userinfo, "user":user})    #⑤
    return render(request, "article/list/article_titles.html",
                  {"articles":articles, "page": current_page})
```

语句④用于创建该用户的 UserInfo 对象实例。读者是否还记得我们在编写关于用户信息的数据模型时，创建了 UserInfo 数据模型类，并且跟 User 类建立了“一对一”的关系（阅读./account/modules.py 文件）。要通过 User 类的实例得到 UserInfo 类的信息，可以使用类似 user.userinfo 的模式，其中 user 是 User 类的实例，而 userinfo 就是 UserInfo，只不过这里一定要用英文小写，这是规定。

在这里使用了 try … except …，主要原因是有的用户可能没有填写自己的个人信息。

语句⑤则是当判断要访问某个用户的文章标题列表时，将 userinfo 也传到模板文件里。

接下来创建一个新的模板文件。在./templates/article/list 目录中创建名为 author_articles.html 的文件，并输入如下代码。

```
{% extends "base.html" %}

{% load staticfiles %}

{% block title %}articles{% endblock %}

{% block content %}
<div class="row text-center vertical-middle-sm">
    <h1>阅读，丰富头脑，善化行为</h1>
</div>
<div class="container">
<div class="col-md-8">
    {% for article in articles %}
    <div class="list-group">
        <a href="{{article.get_url_path}}" class="list-group-item active">
          <h4 class="list-group-item-heading">{{article.title}}</h4>
        </a>
          <p class="list-group-item-text">作者：
             <a href="{% url 'article:author_articles' article.author.username %}">
```

```
                {{article.author.username}}
            </a>
        </p>
        <p class="list-group-item-text">概要:
            {{article.body|slice:'70'|linebreaks}}
        </p>
    </div>
    {% endfor %}
{% include "paginator.html" %}
</div>
<div class="col-md-4">
<div>
    {% if userinfo.photo %}
    <img src="{{ userinfo.photo | striptags }}" class="img-circle" id="my_photo"
         name="user_face" style="width: 300px">
    {% else %}
    <img name="user_face" src="{% static 'images/newton.jpg' %}" class="img-circle"
         id="my_photo" style="width: 300px">
    {% endif %}
</div>
<div>
    <p>{{ user.username }}</p>
{% if userinfo %}
    <p>{{ userinfo.company }}</p>
    <p>{{ userinfo.aboutme }}</p>
{% else %}
    <p>这个作者太懒了，什么也没有留下。</p>
{% endif %}
</div>
</div>
</div>
{% endblock %}
```

在文章标题列表页面的右边，显示该作者的某些信息，相关代码与作者的个人信息代码相似，不再解释，相信读者能够理解。

最后还要看一下效果，如图 4-8 所示。读者是否实现了？

图 4-8　带有作者信息的列表页

4.2.2 知识点

1．数据库简介

虽然本书不讲述有关数据库的操作，但是作为一名网站开发程序员，必须要了解数据库知识，不要认为在 Django 中通过 ORM 操作数据库就足够了，这仅适用于某些项目，可能在有其他要求的项目上，就不能使用 ORM 了，因此必须了解数据库并使用 SQL。

（1）SQLite

SQLite（http://www.sqlite.org/）是一个轻型数据库，它不需要独立的数据库服务器，可以在所有主要的操作系统上运行。在本书的实例中就采用了这种数据库。读者可以查询项目根目录，它的操作与文件操作一样。

虽然 SQLite 比较袖珍，但可以支持高达 2TB 大小的数据库，每个数据库都是以单个文件的形式存在的，这些数据以 B-Tree 数据结构形式存储在磁盘上。尽管如此，读者也不要在真实的项目中使用它，除非保存的数据确实很少。

（2）PostgreSQL

PostgreSQL（https://www.postgresql.org/）号称是“世界上最先进的开源数据库”，这毫不夸张，实至名归。之所以如此，首先，因为它是开源的，曾经还有一个号称开源的 MySQL，但“它变了”，现在只有 PostgreSQL 能扛起开源的大旗了。其次，PostgreSQL 的确有它的特点，比如它的数据类型非常丰富，很多数据库都比不上它；它有专门的 IP 类型，别的数据库没有。再次，也是最值得一提的，PostgreSQL 拥有一支非常活跃的开发队伍，目前的提交人员已经超过 30 人，特别是有众多 hacker 的积极参与（hacker 不是在网上做坏事的人，他们是拥有很好技术的人）。PostgreSQL 的稳定性非常高，受到了开发者的青睐。“稳定压倒一切”，是我们选择数据库的一个重要标准。当然，一般数据库具有的功能 PostgreSQL 一样也不少，比如适合众多操作系统、有众多接口等。

总之，PostgreSQL 值得学习研究，更值得拥有。

（3）MySQL

MySQL（https://www.mysql.com/）曾经是做网站开发的程序员最喜欢的数据库，虽然现在仍有很多网站在使用它，但毕竟是属于 Oracle 旗下的产品了，不再是自由软件了。目前，Oracle 采用了双授权政策，将 MySQL 分为社区版和商业版，一些中小网站也在使用它。MySQL 有着很多优点，所以在网站开发中依然有着举足轻重的地位，特别是在 Linux 作为操作系统、Apache 或 Nginx 作为 Web 服务器、MySQL 作为数据库、PHP/Perl/Python 作为服务器端脚本解释器这种绝配组合中，地位相当稳固。

2．数据库设置

本书中示例的项目采用了 Django 项目创建之后的默认数据库 SQLite，除这个数据库外，Django 项目还支持使用常见的 PostgreSQL、MySQL 和 Oracle。另外，通过第三方模块，还可以支持 IBM DB2、Microsoft SQL Server 等。

数据库的位置在 settings.py（在本书的项目中，位置在./mysite/settings.py），设置 DATABASES

变量的值即可。下面是默认的 SQLite 数据库配置。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

如果是 PostgreSQL 数据库，则可以参考如下配置方式。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql', #数据库服务器
        'NAME': 'mydatabase',    #自己为数据库确定一个名称
        'USER': 'mydatabaseuser', #登录数据库的用户名
        'PASSWORD': 'mypassword',  #上述用户登录数据库的密码
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

3．文档导读

（1）*Databases*，https://docs.djangoproject.com/en/2.1/ref/databases/。

（2）*Multiple databases*，https://docs.djangoproject.com/en/2.1/topics/db/multi-db/。

4.3　为文章点赞

“点赞”这个词语不知道从什么时候开始逐渐替代了“赞赏”“称赞”等词汇，下面就实现对文章点赞的功能，如图 4-9 所示。

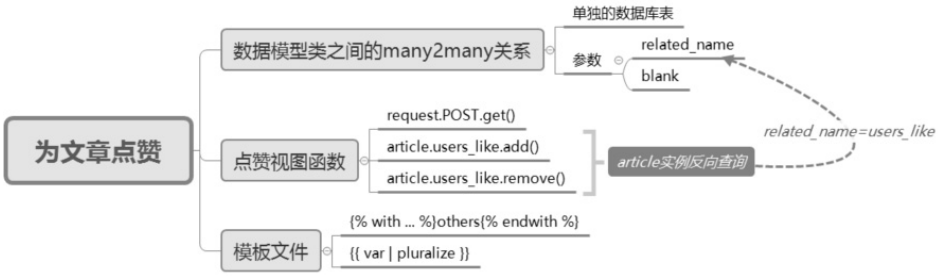

图 4-9　知识技能导图

4.3.1　修改数据模型类

编辑熟悉的 ArticlePost 数据模型类，它在./article/models.py 文件中，在其众多字段后增加如下代码。

```
users_like = models.ManyToManyField(User,
                                     related_name="articles_like", blank=True) #①
```

请读者关注语句①中的一个新的字段类型 ManyToManyField。在本书前面的章节中，已经出现过 OneToOneField 和 ForeignKey 两种字段类型。以上三个字段类型都是用于定义具有关联关系字段的，但它们之间有所差别，具体内容可以参阅 4.3.4 节了解。

很显然，一篇文章可以被多个用户阅读并点赞，而每个用户可以给多篇文章点赞，这就是所谓的“多对多”关系。

将语句①添加到文件中后，要及时完成数据迁移，即执行 python manange.py makemigrations 和 python manage.py migrate。然后看一看数据库表：

```
sqlite> .tables
account_userinfo                auth_user
account_userprofile             auth_user_groups
article_articlecolumn           auth_user_user_permissions
article_articlepost             blog_blogarticles
article_articlepost_users_like  django_admin_log
auth_group                      django_content_type
auth_group_permissions          django_migrations
auth_permission                 django_session
```

其中数据库表 article_articlepost_users_like 就是因为语句①而新增加的（并没有像 ArticlePost 数据模型类中的其他属性那样，在 article_articlepost 数据库表中增加一个 users_like 字段）。下面呈现了这个数据库表的基本结构。

```
sqlite> pragma table_info(article_articlepost_users_like);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           articlepos  integer     1                       0
2           user_id     integer     1                       0
```

可以看到，在这个数据库表中有两个字段：articlepost_id 和 user_id，这说明语句①的作用就是建立起 ArticlePost 和 User 两个数据模型（或者 article_articlepost 和 auth_user 两个数据库表）之间的对应关系。

4.3.2 编写视图函数

数据模型确定好之后，就有了保存数据的数据库表了，接着编写视图函数，完成用户的操作行为。

在./article/list_views.py 文件中引入下列代码的内容。

```
from django.http import HttpResponse
from django.contrib.auth.decorators import login_required
from django.views.decorators.csrf import csrf_exempt
from django.views.decorators.http import require_POST
```

上述后三个引入的都是我们熟悉的装饰器，不再解释，接下来编写如下代码。

```
@csrf_exempt
@require_POST
@login_required(login_url='/account/login/')
def like_article(request):
```

```
    article_id = request.POST.get("id")    #①
    action = request.POST.get("action")    #②
    if article_id and action:
        try:
            article = ArticlePost.objects.get(id=article_id)
            if action=="like":
                article.users_like.add(request.user)    #③
                return HttpResponse("1")
            else:
                article.users_like.remove(request.user)    #④
                return HttpResponse("2")
        except:
            return HttpResponse("no")
```

语句①和语句②得到前端以 POST 方式传过来的两个值，显然这里已经规定好了前端必须用类字典的形式向视图函数提交 id 和 action 的值。

语句③用于给 article 这个实例的属性 users_like 增加一个用户，即该用户“点赞”了此文章。如果不同对象之间建立了一对多或者多对多的关联关系，那么就可以使用 add(*objs, bulk=True)方法增加属性的值，从而建立起两个对象的关系。例如语句③中的 add()方法，就将 article 实例对象和 user 实例对象之间建立了关联，如果这一步成功执行，那么在数据库表 article_articlepost_users_like 中就能够看到 article 和 user 各自的 id 值。以上是多对多关系，如果是一对多（ForeignKey）关系，同样可以使用 add()方法，在 Django 的官方文档中有一个举例，可以赏析（https://docs.djangoproject.com/en/2.1/ref/models/relations/#django.db.models.fields.related.RelatedManager.add）。

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e)    # Associates Entry e with Blog b.
```

语句④是与语句③相反的操作，其使用方法和 add()类似，即把相关联的对象的关系删除。

完成了视图函数的编写，就可以在./article/urls.py 中配置 URL 了，增加如下代码。

```
path('like-article/', list_views.like_article, name="like_article"),
```

4.3.3　修改模板文件

不需要重新编写模板文件，只要在原来展示文章详细内容的模板中增加“点赞”相关的内容即可。./templates/article/list/article_content.html 文件的全部代码如下。

```
{% extends "base.html" %}
{% load staticfiles %}
{% block title %}{{ article.title }}{% endblock %}
{% block content %}

{%  with  total_likes=article.users_like.count  users_like=article.users_like.all  %}
#①

<div class="container">
   <header>
```

```
        <h1>{{ article.title }}</h1>
        <p>
        <a href="{% url 'article:author_articles' article.author.username %}">
            {{ article.author.username }}
        </a>
        <span style="margin-left:20px" class="glyphicon glyphicon-thumbs-up">
            {{ total_likes }}like{{ total_likes | pluralize }}
        </span>    #②
    </p>
    </header>

    <link rel="stylesheet" href="{% static 'editor/css/editormd.preview.css' %}" />
    <div id='editormd-view'>
        <textarea id="append-test" style="display:none;">
{{ article.body }}
         </textarea>
    </div>
    <div>
        <p class="text-center">
            <a onclick="like_article({{article.id}}, 'like')" href="#">
                <span class="glyphicon glyphicon-thumbs-up">like</span>
            </a>    #③
            <a onclick="like_article({{article.id}}, 'unlike')" href="#">
                <span style="margin-left: 15px;"
                      class="glyphicon glyphicon-thumbs-down">unlike</span>
            </a>    #④
        </p>
    </div>
    <div>
        <p class="text-center"><strong>点赞本文的读者</strong></p>
        {% for user in article.users_like.all %}    #⑤
        <p class="text-center">{{ user.username }}</p>
        {% empty %}
        <p class="text-center">还没有人对此文章表态</p>
        {% endfor %}
    </div>
</div>
<script src='{% static "js/jquery.js" %}'></script>
<script src='{% static "editor/lib/marked.min.js" %}'></script>
<script src='{% static "editor/lib/prettify.min.js" %}'></script>
<script src='{% static "editor/lib/raphael.min.js" %}'></script>
<script src='{% static "editor/lib/underscore.min.js" %}'></script>
<script src='{% static "editor/lib/sequence-diagram.min.js" %}'></script>
<script src='{% static "editor/lib/flowchart.min.js" %}''></script>
<script src='{% static "editor/lib/jquery.flowchart.min.js" %}'></script>
<script src='{% static "editor/editormd.js" %}'></script>

<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>  #⑥

<script type="text/javascript">
$(function(){
```

```
    editormd.markdownToHTML("editormd-view", {
       htmlDecode      : "style,script,iframe",  // you can filter tags decode
       emoji           : true,
       taskList        : true,
       tex             : true,  // 默认不解析
       flowChart       : true,  // 默认不解析
       sequenceDiagram : true,  // 默认不解析
    });
});

function like_article(id,action){    #⑦
    $.ajax({
        url: "{% url 'article:like_article' %}",
        type: "POST",
        data: {"id":id, "action":action},
        success: function(e){
            if(e=="1"){
                layer.msg("感谢点赞");
                window.location.reload();
            }else{
                layer.msg("我会继续努力");
                window.location.reload();
            }
        },
    });
}
</script>
{% endwith %}
{% endblock %}
```

语句①包含两部分，一部分是{% with … %}，另一部分是结尾的{% endwith %}，这里使用 with 发起了一个赋值操作，total_likes=article.users_like.count 和 users_like=article.users_like.all 分别为点赞用户总数和用户对象。这样，在两部分所圈定的区间中就可以使用变量 total_likes 和 users_like。请读者学习并掌握这种模板语法。

语句②中增加了{{ total_ likes }}like{{ total_likes | pluralize }}，需要解释的是{{ total_likes | pluralize }}。当没有用户“点赞”或者超过一个用户“点赞”时，显示的是 likes 这种复数形式；如果只有一个用户“点赞”，则显示单数形式 like。这就是{{ total_likes | pluralize }}的作用。

语句③和语句④都是新增的内容，在文章内容的末尾增加“点赞”的操作，而“点赞”功能的实现则是通过 JavaScript 函数（语句⑦）来实现的。提醒读者小心的是，在语句⑦中我们又使用了前面熟悉的弹出提示框的插件，所以要增加语句⑥。

语句⑤是另外一种模板语法的应用。对于 for 循环，我们已经不陌生了。如果循环对象是空（bool()的值为 False），则可能在循环之前用 if 来判断，这里使用{% empty %}，当然，不要忘记最后用{% endfor %}结束，也就是我们在这里使用了{% for … %} … {% empty %} … {% endfor %}的语法结构（前面用过了，是否还有印象？需要反复练习，才能熟知）。

“点赞”功能完成，看看效果吧。

图 4-10 所示为具有点赞功能的界面。

图 4-10　具有点赞功能的界面

请读者注意观察上图中的 likes 目前是复数形式，当用户“点赞”这篇文章之后（刷新），效果如图 4-11 所示。

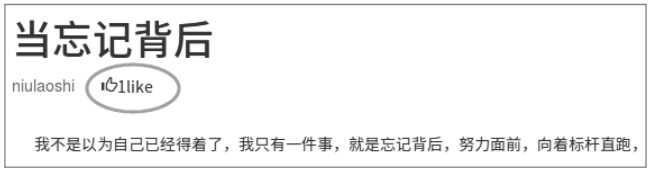

图 4-11　用户“点赞”的效果

请仔细观察，哪些地方发生变化了？

下面看一下数据库中的存储结果：

```
sqlite> select * from article_articlepost_users_like;
id          articlepost_id  user_id
----------  --------------  ----------
1           4               9
2           7               8
3           7               9
```

数据库中记录了对两篇文章的操作的数据。其中 articlepost_id 为 4 的文章得到了 user_id 为 9 的用户点赞，结果如图 4-11 所示；而 articlepost_id 为 7 的文章得到了 user_id 为 8 和 9 的两个用户点赞。

对这个“点赞”功能是不是也“点赞”呢？

4.3.4　知识点

1．模型：“多对多”

本节在 ArticlePost 类中增加了一个字段（users_like = models.ManyToManyField(User, related_name="articles_like", blank=True)），使 ArticlePost 和 User 之间建立了“多对多（ManyToMany）”关系。每篇文章可以被多个用户点赞，而每个用户可以对多篇文章点赞，这就是 ArticlePost 和 User 之间的关系，这种关系被称为“多对多”关系。它跟以前遇到的“一对多”关系一样，也是两个数据模型类之间的关系。

要将两个数据模型类建立“多对多”的关系，只需要在其中一个类中声明一个字段即可，ORM 会自动为另一个类生成使用这种关系的必要的方法和属性（生成一个_set manager 对象）。又因为这种“多对多”是对称关系的，所以在哪一个类中声明都可以。例如在本节中，是在 ArticlePost 中声明了 users_like。

在 Django 的交互模式中，可以看看这种关系的基本特点，代码如下。

```
>>> from django.contrib.auth.models import User
>>> from ArticleMange.models import ArticlePost
>>> article = ArticlePost.objects.get(id=14)
>>> article
<ArticlePost: 伽利略·伽利雷>
>>> one_user = User.objects.get(id=9)
>>> one_user
<User: hulaoshi>
```

建立了 ArticlePost 和 User 两个实例，下面让 one_user 对 article 实例点赞，代码如下。

```
>>> article.users_like.add(one_user)
>>> article.users_like.all()
<QuerySet [<User: yinlaoshi>, <User: hulaoshi>]>
>>> one_user.articles_like.all()
<QuerySet [<ArticlePost: 伽利略·伽利雷>]>
```

article.users_like.all()查询所有对本 article 实例点赞的 User 实例（对某文章点赞的所有用户）；one_user.articles_like.all()查询本 one_user 实例所点赞的 ArticlePost 实例（某用户点赞过的所有文章），请注意，因为有 users_like = models.ManyToManyField(User, related_name="articles_like", blank=True)中的 related_name 的定义，才能使用 one_user.article_like.all()。

“多对多”关系建立之后，在数据库中建立了名为“article_articlepost_users_like”的数据库表，数据库表中保存的是双方的 id。

2．模板：with

with 是模板的一个内置标签，它的使用方法类似于赋值语句，例如本节中的实例。

```
{% with total_likes=article.users_like.count users_like=article.users_like.all %}
<!--{{ total_likes }}相关代码-->
{% endwith %}
```

在变量中使用 article.users_like.count，其拼写太长，不方便，不如使用{{ totla_likes }}更方便，这就如同在 Python 中把一个对象赋值给一个简单的变量，更便于使用。注意，这个变量的有效范围在{% with %}和{% endwith %}之间。

3．模板：for … empty

前面对模板中的 for 标签已经有所介绍，现在补充介绍 for … empty 标签。有时候我们会遇到下面的代码（以下实例来自官网）。

```
<ul>
  {% if athlete_list %}
    {% for athlete in athlete_list %}
      <li>{{ athlete.name }}</li>
    {% endfor %}
  {% else %}
    <li>Sorry, no athletes in this list.</li>
  {% endif %}
</ul>
```

用 if 来判断某个变量是否为空，如果不为空，就循环。因为这种判断在模板中经常会遇到，所以 Django 提供了一种名为 for…empty 的标签，它的使用方法如下。

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% empty %}
    <li>Sorry, no athletes in this list.</li>
{% endfor %}
</ul>
```

4．文档导读

（1）*Many-to-many relationships*，https://docs.djangoproject.com/en/2.0/topics/db/examples/many_to_many。

（2）*django-like*，https://github.com/goinnn/django-like。

4.4 文章的阅读次数

在诸多文章中，哪篇文章最受读者关注？要解决这个问题，就需要记录文章被阅读的次数，本节就学习此功能。下面首先了解一个此前没有接触过的数据库——Redis，维基百科中对此数据库做了如下描述。

> Redis 是一个开源、支持网络、基于内存、键值对存储数据库，使用 ANSI C 语言编写。从 2015 年 6 月开始，Redis 的开发工作由 Redis Labs 赞助。在 2013 年 5 月至 2015 年 6 月期间，其开发工作由 Pivotal 赞助。在 2013 年 5 月之前，其开发工作由 VMware 赞助。根据月度排行网站 DB-Engines.com 的数据显示，Redis 是最流行的键值对存储数据库之一。

本书不对此数据库做详细阐述，读者要深入了解，可以访问其官方网站 https://redis.io/，它还有一个中文网站 http://redis.cn/，或者使用其他相关资料学习。

下面要探讨的是在 Django 中使用 Redis 数据库，实现记录文章阅读次数的功能，如图 4-12 所示。

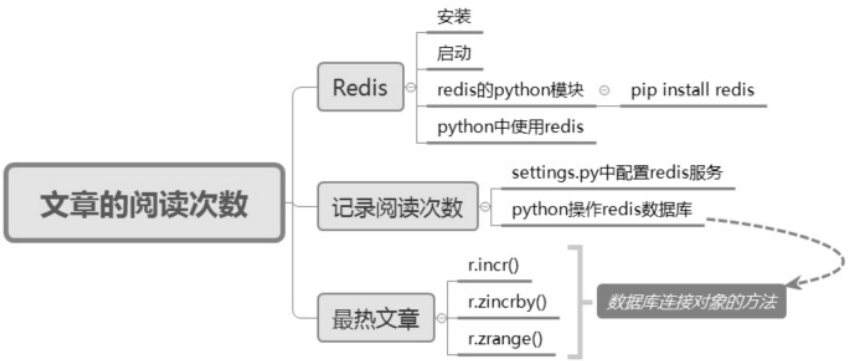

图 4-12　知识技能导图

4.4.1　安装 Redis

笔者写作本书时，Redis 官方发布的最新稳定版是 3.2.6，因此下面的实例就使用此版本。读者阅读本书时，可能会有新版本，也可以下载安装新版本。

安装 Redis 的方法和过程如下。

```
#下载安装包
Chees-MBP:Downloads qiwsir$ wget http://download.redis.io/releases/redis-3.2.6.tar.gz

#解压缩安装包
Chees-MBP:Downloads qiwsir$ tar xzf redis-3.2.6.tar.gz

#将解压缩之后的安装包移动到某个指定目录（此处是 Mac OS 操作系统下的演示），然后进入该目录中
Chees-MBP:Downloads qiwsir$ sudo mv redis-3.2.6/ /usr/local/.
Chees-MBP:Downloads qiwsir$ cd /usr/local/
Chees-MBP:Downloads qiwsir $ cd /usr/local/redis-3.2.6/

#编译安装
$ sudo make install
```

执行编译安装命令后，在终端会显示编译过程，这个过程不用读者干预。等编译完成，就可以进入当前目录下的 src 目录中。

```
$ cd src
$ ./redis-server
```

启动 redis 服务，计算机显示如图 4-13 所示。

在这里我们使用的是默认配置，如果需要修改配置，可以修改 redis.conf 文件（在 redis-.2.6 目录中）。

服务启动之后，再打开一个终端，进入 Redis 环境中，尝试相关操作（以下演示是在本地计算机中的 Redis 安装位置）。

```
Chees-MacBook-Pro:src qiwsir$ pwd
/usr/local/redis-3.2.6/src
Chees-MacBook-Pro:src qiwsir$ ./redis-cli
```

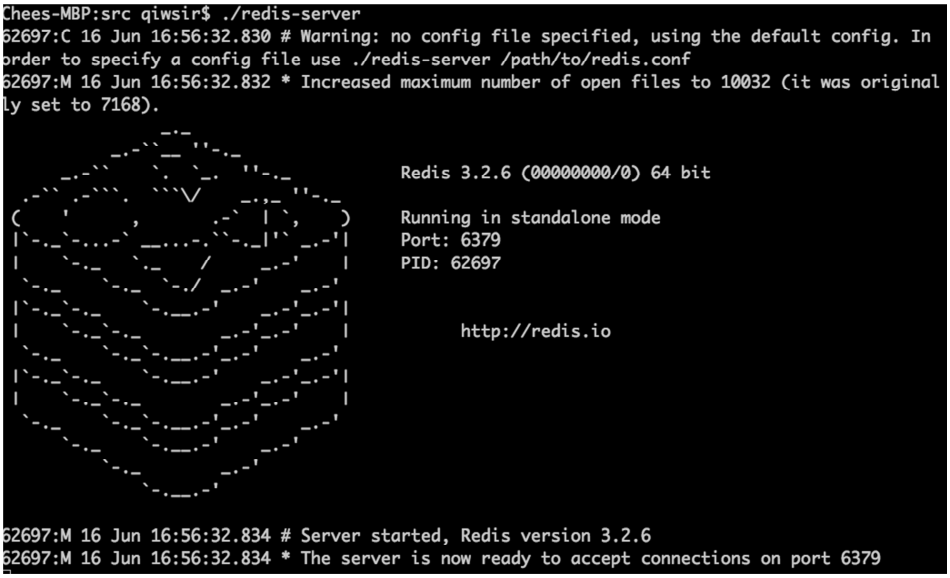
Chees-MBP:src qiwsir$ ./redis-server
62697:C 16 Jun 16:56:32.830 # Warning: no config file specified, using the default config. In order to specify a config file use ./redis-server /path/to/redis.conf
62697:M 16 Jun 16:56:32.832 * Increased maximum number of open files to 10032 (it was originally set to 7168).
Redis 3.2.6 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 62697
http://redis.io
62697:M 16 Jun 16:56:32.834 # Server started, Redis version 3.2.6
62697:M 16 Jun 16:56:32.834 * The server is now ready to accept connections on port 6379

图 4-13　Redis 启动页面

进入 Redis 客户端，输入如下代码，小试牛刀，初步体会 Redis。

```
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> get foo
"bar"
127.0.0.1:6379>
```

如果读者想深入学习 Redis，可以搜集有关资料，此处不再赘述。

4.4.2　在 Python 中使用 Redis

为了能够在 Python 中使用 Redis，还需要安装 Redis 库，安装方法就是使用我们熟悉的 pip，代码如下。

```
$ pip3 install redis
```

安装完成之后，保持 Redis 服务在启动状态下，在新的终端进入 Python 交互模式中，完成如下操作，进行初步体验。

```
>>> import redis
>>> r = redis.StrictRedis(host="localhost", port=6379, db=0)
>>> r.set("foo", "bar")
True
>>> r
StrictRedis<ConnectionPool<Connection<host=localhost,port=6379,db=0>>>
>>> r.get("foo")
b'bar'
```

以上操作都是为了在 Django 中使用 Redis 做准备，下面就进入本节的重点。

4.4.3　记录阅读次数

首先在./mysite/settings.py 文件中做好数据库的配置，即在文件中增加如下变量。

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0
```

然后编辑./article/list_views.py 文件，建立与 Redis 数据库的连接，即在文件中的视图函数代码之前增加如下代码。

```
import redis
from django.conf import settings    #①

r = redis.StrictRedis(host=settings.REDIS_HOST,
                      port=settings.REDIS_PORT, db=settings.REDIS_DB)
```

语句①用于引入本项目 setting.py 中的变量，从而能够在下一句的数据库连接中使用。请读者注意这种方式，如果在别的应用中，如有必要，可以使用这种方式在 setting.py 中做好变量配置，其配置参数可以在各处使用，如 settings.REDIS_PORT 就是调用了 settings.py 文件中的 REDIS_PORT 的值 6397。这种使用方法的本质就是在 Python 中将.py 文件视为模块（语句①的效果），不论文件是变量、函数还是类，都可以通过该模块调用。

编辑本文件中的 article_detail()视图函数，其修改之后的全部代码如下。

```
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    total_views = r.incr("article:{}:views".format(article.id))      #②
    return render(request, "article/list/article_content.html",
                  {"article":article, "total_views":total_views})    #③
```

语句②是新增加的，其功能就是对访问文章的次数进行记录。这里用到了 incr 函数，为了理解这个函数的使用方法，可以在 Redis 的客户端进行一番操作，代码如下。

```
127.0.0.1:6379> set foo 0    #④
OK
127.0.0.1:6379> incr foo
(integer) 1
127.0.0.1:6379> incr foo
(integer) 2
127.0.0.1:6379> incr foo
(integer) 3
127.0.0.1:6379> get foo
"3"
127.0.0.1:6379> incr foo
(integer) 4
127.0.0.1:6379> get foo
"4"
```

显然，incr 函数的作用就是让当前的键值递增，并返回递增后的值。在语句④中首先设置键 foo 的值为零，然后进行递增。

Redis 对于键的命名并没有强制的要求，比较好的做法是用“对象类型:对象 ID:对象属性”

来命名一个键，比如语句②中使用键 article:15:views 来存储 ID 为 15 的文章的浏览次数。

在语句③中将文章访问次数传给模板文件，接下来就要在模板文件中展示计数结果。

编辑./templates/article/list/article_content.html 文件，找到原来显示点赞数量的代码，在那里增加访问次数的代码（下述代码中的语句⑤是针对访问次数新增的部分）。

```
<p>
<a href="{% url 'article:author_articles' article.author.username %}">
   {{ article.author.username }}
</a>
<span style="margin-left:20px" class="glyphicon glyphicon-thumbs-up">
   {{ total_likes }}like{{ total_likes | pluralize }}
</span>
<span style="margin-left: 20px">
   {{ total_views }}view{{ total_views | pluralize }}
</span>    #⑤
</p>
```

启动 Django 服务，访问任何一篇文章并查看访问次数，然后刷新该文章，再次观察访问次数，如图 4-14 所示。

图 4-14　记录文章的浏览次数

已经实现了文章访问次数记录的功能。那么，能不能将访问次数最多的文章排列到一个“最热文章”的栏目里面呢？需求层出不穷，只有这样才能提高自己的技能。

4.4.4　显示最“热”文章

在刚刚实现的文章访问计数的基础上，实现显示“最热文章”的功能。继续编辑./article/list_views.py 文件中的视图函数 article_detail()，增加一些代码。为了能够让读者清晰阅读，下面把此函数的全部代码列出来。

```
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    total_views = r.incr("article:{}:views".format(article.id))
    r.zincrby('article_ranking', 1, article.id)    #①

    article_ranking = r.zrange('article_ranking', 0, -1, desc=True)[:10]    #②
    article_ranking_ids = [int(id) for id in article_ranking]
    most_viewed = list(ArticlePost.objects.filter(id__in=article_ranking_ids))    #③
    most_viewed.sort(key=lambda x: article_ranking_ids.index(x.id))    #④
```

```
    return render(request, "article/list/article_content.html",
                  {"article":article,
                   "total_views":total_views,
                   "most_viewed": most_viewed})    #⑤
```

语句①中使用了 Redis 连接对象的一个方法 zincrby（要了解这个方法的具体含义，可以在 Python 的交互模式中使用 help()命令，这种自省方法在《跟老齐学 Python：轻松入门》一书中反复强调，是读者学习 Python 的最好方法），完整形式是 zincrby(name, amount, value)，其作用是根据 amount 所设定的步长值增加有序集合（name）中的 value 的数值。在语句①中实现了 aticle_ranking 中的 article.id 以步长 1 自增，即文章被访问一次，article_ranking 就将该文章 id 的值增加 1。

语句②承接语句①的结果，得到 article_ranking 中排序前 10 名的对象。zrange(name, start, end, desc=False, withscores=False, score_cast_func=float)也是数据库连接对象的一个方法，读者从参数的名称中也能理解该方法的基本调用形式。

在上述操作的基础上，语句③得到最“热”文章对象组成的列表。注意“id__in”中间是双下画线，这是 Django 里面的一种相当于条件查询的语句，其功能是查询出 id 在 article_ranking_ids 中的所有文章对象，并以文章对象为元素生成列表。

语句④是对得到的列表进行排序。请读者注意排序关键词条件，使用了 lambda 函数（如果读者对此函数不了解，请参阅《跟老齐学 Python：轻松入门》的有关章节，对此函数有专门讲解）。

语句⑤中的"most_viewed": most_viewed 相对未修改之前是新增的，请注意不要丢掉。

完成视图函数代码的修改后，再修改模板文件./templates/article/list/article_content.html，这是文章详细内容页面，我们要实现的功能是将页面分为左右两部分，左边依然显示文章的完整内容，右边则用于显示刚才所设定的“最热文章”栏目。

先将原有的文章内容 HTML 代码 包裹起来，然后创建展现“最热文章”栏目的代码块 ，详细代码如下。

```
#省略原有部分的代码
<div class="container">
<div class="col-md-9">
   #省略显示文章详细内容部分的代码
</div>
<div class="col-md-3">
<p class="text-center"><h3>最受欢迎文章</h3></p>
<ol>
   {% for article_rank in most_viewed %}
   <li>
      <a href="{{article_rank.get_url_path}}">{{ article_rank.title }}</a>
   </li>
   {% endfor %}
</ol>

</div>
</div>
#省略下面的 JavaScript 代码
```

如果读者阅读上述代码有困难，请复习本书前面的内容，这里不再赘述。最终效果如图 4-15 所示（在查看最终效果前，请确认 Redis 数据库的服务和 Django 服务已经启动）。

本节我们用 Redis 实现了最“热”文章排序。其实，在 Django 中实现类似功能还有别的方法，后面会陆续讲解。

图 4-15　显示最受欢迎文章

Redis 的用途绝非上面所讲述的这些，在某些项目中，它的作用非常大，比如某博网站，据说在众多业务场景中使用了 Redis，如关系、计数、通知提醒等，每天提供了上万亿次的读取访问。

至此，读者是不是已经感受到了一点点挑战？千万不要放弃，“行百里者半九十”，坚持就是胜利。如果你反复调试还是没有解决 Bug，有点沮丧甚至畏惧，那么请阅读下面的歌词。

你是不是像我曾经茫然失措

一次一次徘徊在十字街头

因为我不在乎别人怎么说

我从来没有忘记

我对自己的承诺，对爱的执着

我知道我的未来不是梦

我认真地过每一分钟

我的心跟着希望在动

歌曲《我的未来不是梦》献给每一位在代码海洋中畅游的朋友。

4.4.5　知识点

1．关于 Redis

本节以最简单的方式使用了 Redis，实际上在一些大型的网站中，Redis 应用广泛。下面首先介绍它的基本特点。

- Redis 将其数据库完全保存在内存中，仅使用磁盘进行持久化。
- 与其他键值数据存储相比，Redis 有一组相对丰富的数据类型。
- Redis 可以将数据复制到任意数量的其他从属服务器。

上述特点使得 Redis 具有以下优势。

- Redis 读取速度非常快：每秒可执行大约 110000 次的设置（SET）操作，每秒大约可执行 81000 次的读取/获取（GET）操作。
- 支持丰富的数据类型：Redis 支持常用的大多数数据类型，例如列表、集合、排序集和散列等。
- 操作具有原子性：所有 Redis 操作都是原子操作，如果两个客户端并发访问，那么 Redis 服务器接收更新的值。
- 多实用工具：Redis 是一个多实用工具，可用于多种用例，例如缓存、消息队列等。此外，Web 应用程序中的会话、网页访问量计数等也经常使用 Redis。

根据特点和优势，可以想象一下 Redis 的用武之地，比如网站搞活动有限量抢购，如果每次都访问数据库恐怕会对数据库造成很大的压力，这时就可以借助 Redis；本节实例也是一种应用场景。

2．文档导读

（1）django-redis 中文文档，http://django-redis-chs.readthedocs.io/zh_CN/latest/。

（2）*Caching in Django With Redis*，https://realpython.com/blog/python/caching-in-django-with-redis/。

4.5　文章的评论功能

除浏览文章外，用户还可以对文章品头论足，发表自己的观点。本节需要实现的评论功能包括：

- 登录网站的用户有权限对任何文章发表评论；
- 评论内容都显示在文章的后面。

文章的评论功能如图 4-16 所示。

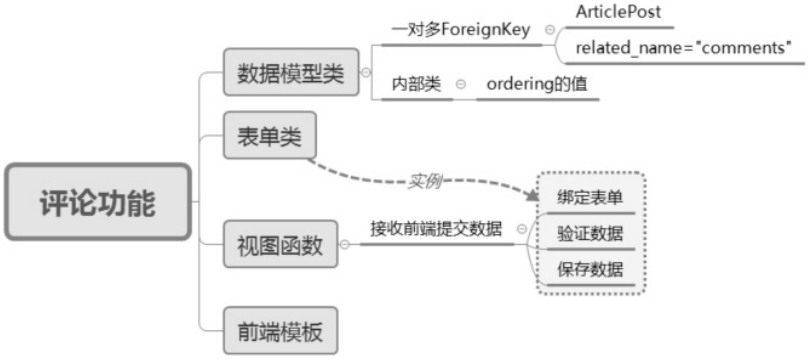

图 4-16　知识技能导图

4.5.1　数据模型类和表单类

为了实现评论功能，需要建立相应的数据库表，用来存储评论内容。在./article/models.py 文件中创建与评论功能相关的数据模型类并编辑这个文件，代码如下。

```
class Comment(models.Model):
    article = models.ForeignKey(ArticlePost, on_delete=models.CASCADE,
                                related_name="comments")    #①
    commentator = models.CharField(max_length=90)
    body = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('-created',)    #②

    def __str__(self):
        return "Comment by {0} on {1}".format(self.commentator.username, self.article)
```

在语句①中通过 ForeignKey()将本数据模型与 ArticlePost 建立关系，它们属于多对一的关系，即一篇文章可以有多篇评论。

语句②是容易出错的地方，请读者务必注意这里使用的是元组类型，不要丢掉后面的逗号；负号意味着按照 created 的倒序排列。

数据模型类编写好之后，就可以实施同步数据库的操作了（依次执行 python manage.py makemigrations 和 python manage.py migrate）。为了满足好奇心，可以查看一下数据库的基本结构。

```
sqlite> .tables
account_userinfo              auth_user
account_userprofile           auth_user_groups
article_articlecolumn         auth_user_user_permissions
article_articlepost           blog_blogarticles
article_articlepost_users_like  django_admin_log
article_comment               django_content_type
auth_group                    django_migrations
auth_group_permissions        django_session
auth_permission
sqlite> pragma table_info(article_comment);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1           1
1           commentato  varchar(90  1           0
2           body        text        1           0
3           created     datetime    1           0
4           article_id  integer     1           0
```

为了能够在文章的后面给用户显示一个填写评论的输入框，还要建立表单类。在./article/forms.py 文件中编写表单类，代码如下。

```
from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ("commentator", "body",)
```

基础工作完成，下面就要实现具体的功能。

4.5.2　实现评论功能

评论功能是在用户阅读文章的页面中，所以需要再次编辑./article/list_views.py 文件中的article_detail()函数。读者可以先看一下这个函数目前的样子，为了实现评论功能，要对此函数增加内容。

首先在./article/list_views.py 文件中引入 Comment 类和 CommentForm 类，代码如下。

```
from .models import Comment
from .forms import CommentForm
```

然后对 article_detail()函数进行重写，下面将重写后的全部代码都呈现出来，便于读者进行比较。

```
def article_detail(request, id, slug):
    article = get_object_or_404(ArticlePost, id=id, slug=slug)
    total_views = r.incr("article:{}:views".format(article.id))
    r.zincrby('article_ranking', 1, article.id)
    article_ranking = r.zrange('article_ranking', 0, -1, desc=True)[:10]
    article_ranking_ids = [int(id) for id in article_ranking]
    most_viewed = list(ArticlePost.objects.filter(id__in=article_ranking_ids))
    most_viewed.sort(key=lambda x: article_ranking_ids.index(x.id))

    if request.method == "POST":    #①
        comment_form = CommentForm(data=request.POST)
        if comment_form.is_valid():
            new_comment = comment_form.save(commit=False)
            new_comment.article = article
            new_comment.save()
    else:
        comment_form = CommentForm()

    return render(request, "article/list/article_content.html",
                  {"article":article, "total_views":total_views,
                   "most_viewed": most_viewed, "comment_form":comment_form})
```

从语句①开始是处理前端以 POST 形式提交过来的表单数据的，并且在验证（comment_form.is_valid()）通过之后，保存到数据库。

编辑模板文件./article/list/article_content.html，在“点赞本文的读者”功能代码块后面增加如下内容。

```
<div class="col-md-9">
#原有代码从略，以下为新增部分
<div>
<p class="text-center"><strong>点赞本文的读者</strong></p>
   {% for user in article.users_like.all %}
   <p class="text-center">{{ user.username }}</p>
   {% empty %}
   <p class="text-center">还没有人对此文章表态</p>
   {% endfor %}
</div>
```

```
<!—以上是原有的，以下是为评论功能增加的代码—>
    <hr>
    <div>
        <h3><span class="glyphicon glyphicon-bullhorn"></span>本文有{{ comments.count }}
评论</h3>
        {% for comment in article.comments.all %}    #②
        <div>
            <p><strong>{{ comment.commentator }}</strong>说：</p>
            <p style="margin-left:40px;">{{ comment.body }}</p>
        </div>
        {% empty %}
        <p>没有评论</p>
        {% endfor %}

        <h3><span class="glyphicon glyphicon-send"></span>看文章，发评论，不要沉默</h3>
        <form   action="."   method="post"   class="form-horizontal"   role="form">{%
csrf_token %}
            <div class="form-group">
                <label for="inputEmail3" class="col-sm-2 control-label">评论员</label>
                <div class="col-sm-10">
                    {{ comment_form.commentator}}
                </div>
            </div>
            <div class="form-group">
                <label for="inputEmail3" class="col-sm-2 control-label">评论</label>
                <div class="col-sm-10">
                    {{ comment_form.body }}
                </div>
            </div>
            <div class="form-group">
                <div class="col-sm-offset-2 col-sm-10">
                    <p><input  type="submit"  name=""  value=" 发 评 论 "  class="btn
btn-primary"></p>
                </div>
            </div>
        </form>
    </div>
</div>
```

上述新增的代码主要有两个功能，一个是显示已有的评论，另一个是显示发表评论的表单。

语句②中以 article.comments.all 得到文章的所有评论，请读者注意这种模板语言的用法，类似于 Django 的 QuerySet 语句。

此处，当用户提交表单时，没有使用我们熟悉的 Ajax 方式，而是将<input>的类型设置为 submit，可以直接提交到所规定的 URL。这种方式虽然简单，但也存在瑕疵。读者不妨在调试中观察。当然，特别鼓励读者将这里修改为 Ajax 方式，具体修改方法可以参照本书前面的有关章节内容。

模板文件编写好之后，就可以看演示效果了。不过，要提醒读者注意的是，因为我们前面使用了 Redis 数据库，要保证它处于运行状态，并且确保 Djano 服务也启动了。浏览某篇文章，

可以在底部进行评论，如图 4-17 所示。

图 4-17　评论功能

这就是评论功能。

4.5.3　知识点

1．模型：Meta

在数据模型类的内部，有时候会定义 class Meta，这个类的名字很有特点，读者如果熟悉 Python，就会想到在 Python 中有 metaclass（元类），但在数据模型类内部定义的 Meta 和 Python 中的 metaclass 是有区别的，我们把在数据模型类内部定义的名为 Meta 的类称为内部类。

内部类的作用是什么？在 Python 中，类的内部以 class Meta 方式来写一个内部类，其主要作用是让该类的不同实例共用一个属性值。为了便于理解，请看下面的实例。

```
>>> class Foo:    #①
...     class Meta:
...         name = "canglaoshi"
...
>>> teacher = Foo()
>>> good_teacher = Foo()
>>> teacher.Meta.name    #②
'canglaoshi'
>>> good_teacher.Meta.name = "itdiffer.com"    #③
>>> teacher.Meta.name    #④
'itdiffer.com'
```

语句①定义了一个类 Foo，在其内部有内部类 Meta，这个内部类中只有一个变量 name，并且已经赋值了。然后建立了两个实例 teacher 和 good_teacher，在一般情况下，实例 teacher 的属性值和实例 good_teacher 的属性值是不会互相影响的，而是相对独立的。语句②得到了实例 teacher 的内部类中的属性 name 的值。语句③对另外一个实例 good_teacher 的内部类属性 name 重新赋值。语句④将该值共享给了 teacher 这个实例的内部类的 name 属性。这就是内部类的作用。

Django 数据模型中的内部类也规定了该数据模型不论哪个实例，都应该具有的行为。

在内部类中，通过声明一些属性的值，让该数据模型具有某些指定的行为，比如本节 Comment 类中的代码。

```
class Meta:
    ordering = ('-created',)
```

本来 Comment 的实例（数据库表中的记录）有默认的排序字段，但这里内部类中重新规定了排序字段和排序原则，即按照 created 的值的倒序排列。

内部类中的变量不是数据模型类的字段，官方文档中用“anything that’s not a field”来说明了其意义。

- abstract

用于定义当前的模型类是不是一个抽象类。抽象类不对应数据库表。一般用它来归纳一些公共属性字段，继承它的子类可以继承这些字段。

- app_label

如果当前的数据模型不在 INSTALLED_APPS（settings.py 中）所注册的应用中，那么就需要用 app_label 来声明本数据模型所属于的应用。比如在其他地方写了一个模型类，而这个模型类是属于 myapp 的，就需要设置 app_label='myapp'。

- db_table

用于指定自定义数据库的表名。在默认状态下，完成数据库迁移之后，会按照 Django 的规定生成数据库表名，而这个属性提供了一种自定义表名的途径。

- index_together

以列表类型的值声明索引字段名称，例如 index_together = ["pub_date", "deadline"]。

2．文档导读

（1）*Model Meta options*，https://docs.djangoproject.com/en/2.1/ref/models/options/。

（2）*What is a metaclass in Python*，http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python。

4.6 多样化显示

从功能上看，已经实现了对文章的常规操作。为了让网站功能更丰富，本节将实现另外三个功能，包括：

- 某用户发布的文章总数统计；
- 最新发布的文章列表；
- 评论最多的文章列表。

本来利用前面已经学习的知识是可以实现上述三个功能的（读者如果有兴趣，可以在此暂时中止阅读，利用已经掌握的知识实现上述三个功能），但为了让读者进一步学习 Django，下

面介绍一种新的方式来实现上述功能。

本节要学习的新的方式就是“自定义模板标签”。

对于模板标签，我们已经不陌生了，比如前面广泛使用的{% if %}和{% for %}等，这些都是 Django 模板中默认的模板标签。

除有默认的模板标签外，Django 为了应对更多复杂的操作，还允许开发者自定义模板标签。Django 中一共有三种自定义模板标签类型，分别是 simple_tag、inclusion_tag 和 assignment_tag。下面在实例中依次讲解这三种类型的自定义标签的使用方法，顺便完成前面提到的三个功能，如图 4-18 所示。

图 4-18　知识技能导图

4.6.1　统计文章总数

在./article 中创建一个目录 templatetags，然后在里面创建两个文件__init__.py 和 article_tags.py，如下所示。

```
├──── templatetags
|     ├──── article_tags.py
|     └──── __init__.py
```

目录 templatetags 的名称是 Django 默认的，必须用这个名称，不能更改。article_tags.py 的名称则是笔者自己定义的，习惯上使用类似这样的命名方式，在后面应用的时候也比较方便。在 templatetags 目录中，可以编写多个类似 article_tags.py 的文件，在模板中只要声明使用哪个文件中的标签即可。

下面的工作主要是在 article_tags.py 文件中定义模板标签。

先来编写实现统计文章总数的模板标签，在./article/templatetags/article_tags.py 中增加如下代码。

```
from django import template    #①

register = template.Library()    #②

from article.models import ArticlePost    #③

@register.simple_tag    #④
def total_articles():
    return ArticlePost.objects.count()    #⑤
```

语句①引入了 django.template 库，它里面包含着诸多与模板有关的类和方法等，语句②中的 Library()就是其中一员。语句②创建了一个实例对象 register，这个对象包含了 simple_tag 方法（及另外两种），它将用于自定义标签中。注意这里的变量名称要使用 register，它是 template.Library 的一个实例。

语句③将有关数据模型类引入本文件，请读者注意观察引入方式。

语句④通过装饰器，表明其下面的代码是自定义的 simple_tag 类型的标签。

语句⑤为返回文章对象数量的查询结果。特别要注意，这里不是视图函数，所以不要习惯性地在参数中写 request 了，也没有 render()。

这样我们就定义好了一个标签。下面将这个标签应用到./templates/article/list/article_titles.html 中，编辑此文件，在文件顶部写上{% load article_tags %}。

```
{% extends "base.html" %}
{% load article_tags %}    #⑥
```

语句⑥就是我们要引入自定义标签的声明，article_tags 应该与./article/templatetags 目录中的标签所在的文件名称一致，这样 Django 就知道到该文件中查找所定义的标签（函数）。

在此文件中应用所定义的标签，代码如下。

```
<div class="row text-center vertical-middle-sm">
    <h1>阅读，丰富头脑，善化行为</h1>
    <p>这里已经有{% total_articles %}篇文章供你阅读</p>    ⑦
</div>
```

语句⑦中的{% total_articles %}就是刚刚定义的标签，total_articles 与 article_tags.py 文件中的函数名称一致。在这个位置，就可以显示本网站中一共有多少篇文章。

重启 Django 服务，不要忘记确认 Redis 服务已经运行，访问 http://localhost:8000/article/list-article-titles/，显示效果如图 4-19 所示。

图 4-19　显示文章统计数量

这里仅显示了总的文章数量，能不能显示某个作者一共发布了多少篇文章？

当然可以，还是使用自定义标签的方法。继续在./article/templatetags/article_tags.py 文件中编写 simple_tag 类型的标签函数，代码如下。

```
@register.simple_tag
def author_total_articles(user):    #⑧
    return user.article.count()     #⑨
```

从语句⑨可以看出来，参数 user 应该是一个用户对象，而不是字符串。语句⑨之所以能够以 user.article.count()方式得到该用户的文章总数，是因为在./article/models.py 文件中定义 ArticlePost 数据模型类时，其中属性 author = models.ForeignKey(User, on_delete=models.CASCADE, related_name="article")，又一次体现了 related_name 的作用。

标签定义好之后，编辑模板文件./templates/article/list/author_articles.html，依然要在顶部引入{% load article_tags %}（其实，在下述使用之前引入即可，如果习惯放在顶部，与其他{% load xxx %}放在一起），位置与上面的模板位置一样。

然后在适当位置加入自定义的标签，代码如下。

```
<div>
    <p>{{ user.username }}</p>
    <p>共发表文章{% author_total_articles user %}篇</p>    #⑩
    {% if userinfo %}
    #其他代码从略
</div>
```

语句⑩就是这次增加的显示该作者发表文章数目的标签。重启 Django 服务，访问某作者文章列表页面，查看是否实现了效果，如图 4-20 所示。

图 4-20　统计每个作者的文章数量

从上面的例子中可以看出，simple_tag 类型的标签返回的是字符串。这里仅演示了 simple_tag 的基本引用，官方文档中还有更多的说明（https://docs.djangoproject.com/en/2.1/howto/custom-template-tags/#simple-tags）供读者参考。另外，读者在调试时要注意，当自定义标签后，需要将 Django 服务重新启动。

4.6.2 最新发布的文章

汇总最新发布的文章这个功能也可以通过自定义标签实现，这次使用 inclusion_tag 来实现。

还是在./article/templatetags/article_tags.py 中增加自定义标签的函数，代码如下。

```
@register.inclusion_tag('article/list/latest_articles.html')    #①
def latest_articles(n=5):    #②
    latest_articles = ArticlePost.objects.order_by("-created")[:n]    #③
    return {"latest_articles": latest_articles}    #④
```

语句①仍然使用装饰器来声明自定义的标签类型，只不过这次增加了参数，用('article/list/latest_articles.html')确定所渲染的模板文件。这是因为对于 inclusion_tag 类型的标签，返回语句④样式的字典类型的数据，此数据被应用到语句①中所指定的模板文件中。

语句②定义了函数名称和参数，在具体使用此标签时会向这里传入参数。

语句③中的 order_by()用于实现按照文章发布的时间倒序（"-created"）查询所有文章对象，并且根据参数（n）的值，获取排在前面的若干个（n 个）对象。这里得到的值是一个以文章对象为元素组成的列表。

这个函数写好之后，就在./template/article/list 中创建语句①所指定的模板文件 latest_articles.html，其代码如下。

```
<ul>
   {% for article in latest_articles %}    #⑤
   <li>
    <a href="{{ article.get_url_path }}">{{ article.title }}</a>    #⑥
   </li>
   {% endfor %}
</ul>
```

语句⑤使用自定义标签函数 latest_articles()所返回的字典数据的键（Key）latest_articles，它的值是一个由文章对象组成的列表。

语句⑥中的{{ article.get_url_path }}来源于./article/models.py 文件中 ArticlePost 类里面的 get_url_path()方法。

上述工作完成后，算是写好了一个 inclusion_tag 类型的自定义标签（见语句①）。下面要在./templates/article/list/article_content.html 模板文件中使用这个标签，首先在此文件顶部引入标签{% load article_tags %}。

已经有了“最受欢迎文章”栏目，现在要在其下面显示“最新文章”栏目，于是增加如下代码。

```
<div class="col-md-3">
   <p class="text-center"><h3>最受欢迎文章</h3></p>  #这是原有代码
   #略去原有代码，下面三行是新增的
   <hr>
   <p class="text-center"><h3>最新文章</h3></p>
   {% latest_articles 4 %}    #⑦
</div>
```

语句⑦中使用自定义标签 latest_articles，后面的数字是向这个自定义标签函数传入的参数值，即显示最新的 4 篇文章。

重新启动 Django 服务，然后访问某篇文章，应该出现如图 4-21 所示的效果。

图 4-21　显示最新文章

从上面的过程中不难看出，inclusion_tag 类型的标签实际上是返回一个模板（装饰器函数中指定的模板文件）。当调用该标签名称时，显示标签装饰器函数中所指定的模板文件的显示方式。inclusion_tag 的官方文档是 https://docs.djangoproject.com/en/2.1/howto/custom-template-tags/#inclusion-tags。

4.6.3　评论最多的文章

下面要找出评论最多的文章，并且将标题列出来，还要得到一系列的文章对象。虽然不使用自定义标签也能够实现，但这里为了学习另外一种类型（assignment_tag）的自定义标签，我们还是承接前面的做法，在./article/templatetags/article_tags.py 中编写自定义标签的有关函数，代码如下。

```
from django.db.models import Count
@register.simple_tag    #①
def most_commented_articles(n=3):    #②
    return ArticlePost.objects.annotate(total_comments=Count('comments')).order_by
                                    ("-total_comments")[:n]    #③
```

语句①依然使用装饰器来声明下面的函数是自定义的 simple_tag 类型的标签函数（在 Django 1.x 版本中，语句①用 assignment_tag 类型的标签函数）。

语句②中以参数的形式说明显示评论最多的文章数量，这里默认显示评论数量最多的前 3 篇。

语句③是一条稍微复杂的语句，为了能够清晰地理解，我们可以先打开一个终端，在交互模式中分步研究（面对一个复杂的东西，分解、隔离是一种常用的方法）。

还记得怎样进入 Django 的交互模式吗？进入项目目录，执行 python manage.py shell，即可进入交互模式。

```
>>> from django.db.models import Count
>>> from article.models import ArticlePost
```

```
>>> c = ArticlePost.objects.annotate(total_comments=Count("comments"))    #④
>>> c
<QuerySet [<ArticlePost: 物理学家赫兹>, <ArticlePost: 伽利略·伽利雷>, <ArticlePost: Albert Einstein>, <ArticlePost: Isaac Newton>, <ArticlePost: Heinrich Hertz>, <ArticlePost: You Raise Me Up>]>
```

语句④中 annotate()函数的作用是给 QuerySet 中的每个对象添加注释。如果仅这么说，估计读者会感到迷茫，可是如果去搜索，会发现不少文章中都是这样解释的，更有甚者直接说 annotate()函数的作用就是 annotate（注释）。语言是思维的工具，annotate 的汉语解释一般就是“注释”，但是在这里我们如何描述其作用呢？看本书慢慢道来。

语句④其实还可以继续分解。

```
>>> articles = ArticlePost.objects.all()
>>> articles
<QuerySet [<ArticlePost: 物理学家赫兹>, <ArticlePost: 伽利略·伽利雷>, <ArticlePost: Albert Einstein>, <ArticlePost: Isaac Newton>, <ArticlePost: Heinrich Hertz>, <ArticlePost: You Raise Me Up>]>
```

articles 就是查询到的所有文章对象序列，然后对这个文章序列进行注释。

```
>>> a = articles.annotate(total_comments=Count("comments"))
```

现在是不是已经逐步理解了？annotate()函数是要给查询到的文章对象进行标注。用什么标注呢？就是里面的参数 Count("comments")。

Count("comments")的作用是给每篇文章的评论计数，这里的 comments 来自./article/models.py 文件的 Comment 数据模型类中定义的 article = models.ForeignKey(ArticlePost, on_delete=models.CASCADE, related_name="comments")，related_name 再次实现了通过 ArticlePost 对象查询到其相关的 Comment 对象，所以 Count("comments")得到的是 ArticlePost 对象关联的 Comment 对象的个数，将这个个数标注到前面得到的 articles 所对应的 ArticlePost 对象上，这就是语句④的含义。有点拗口，笔者实在没有能力用别的方式通俗且严谨地表述了，请读者参考，并阅读更多的相关资料，形成自己的理解。

接着对语句④的操作进行探索。

```
>>> c[0].total_comments
5
>>> c[1].total_comments
1
```

既然语句④得到的结果是由一个个 ArticlePost 对象组成的列表，并且每个 ArticlePost 对象都标注上了 total_comments=Count("comments")，那么就可以理解为每个对象都有了 total_comments 属性，其实就是该 ArticlePost 对象所关联的 Comment 对象的数量，即评论的数量。因此，通过 c[0].total_comments 就得到了第一个 ArticlePost 对象的 Comment 对象数量（就是得到第一篇文章的评论数目）。

继续保持慢速咀嚼式的阅读。

```
>>> c.order_by("-total_comments")    #⑤
<QuerySet [<ArticlePost: 物理学家赫兹>, <ArticlePost: 伽利略·伽利雷>, <ArticlePost: You Raise Me Up>, <ArticlePost: Heinrich Hertz>, <ArticlePost: Isaac Newton>, <ArticlePost: Albert Einstein>]>
```

语句⑤就是对上述得到的序列排序，按照 total_comments 数量从大到小进行排序。读者可能发现结果与前面一致，这纯粹是一种巧合，完全可以去掉符号，像下面一样。

```
>>> c.order_by("total_comments")
<QuerySet [<ArticlePost: You Raise Me Up>, <ArticlePost: Heinrich Hertz>, <ArticlePost: Isaac Newton>, <ArticlePost: Albert Einstein>, <ArticlePost: 伽利略·伽利雷>, <ArticlePost: 物理学家赫兹>]>
```

读者不妨将上面的内容反复思考，并且在交互模式中进行尝试。“纸上得来终觉浅，绝知此事要躬行”。

理解了语句③的含义，自定义标签函数就完成了。

编辑./templates/article/list/article_content.html 文件，实现“最多评论文章”栏目的显示。将下面的代码写在“最新文章”栏目的代码块后面。

```
<hr>
<p class="text-center"><h3>最多评论文章</h3></p>
{% most_commented_articles as most_comments %}    #⑥
<ul>
    {% for comment_article in most_comments %}    #⑦
    <li>
        <a href="{{comment_article.get_url_path}}">{{ comment_article.title }}</a>
    </li>
    {% endfor %}
</ul>
```

语句⑥是必需的，即把自定义标签所得到的对象赋值给一个变量，然后在语句⑦中循环此变量。

保存所有修改文件，并且重启 Django 服务。访问某篇文章，查看效果，如图 4-22 所示。

图 4-22　最多评论文章功能

自定义标签能够让我们在前端很方便地实现某些渲染结果。对于前端，还可以做其他自定义的内容。

4.6.4 自定义模板选择器

对于模板选择器，在前面曾经遇到过，比如显示文章前多少个字作为摘要，使用了{{article.body|slice:'70'| linebreaks}}，这就是一个模板选择器（filter 有时也被翻译为“过滤器”）。模板选择器的基本使用方式是{{ variable | filter }}或者{{ variable | filter:"foo" }}。其中，variable 代表传入模板的对象的变量，filter 是所传入的值按照一定的规则进行选择的选择器，foo 则是给选择器传入的参数。还可以同时使用多个选择器，样式为{{ variable | filter1 | filter2 }}。

下面要做一个选择器，用来实现在模板页面将 Markdown 语法解析为 HTML 代码，从而在页面中显示。虽然类似的工作我们在前面已经做过了，但是那时用的是纯粹的前端 JavaScript 插件完成的，而这里要使用自定义的模板选择器。

安装 Markdown 第三方库，这是我们要在项目中使用的。

```
$ pip3 install Markdown
```

然后在./article/templatetags/article_tags.py 文件中编写将 Markdown 编码转换为 HTML 代码的选择器函数。

在文件顶部引入如下内容。

```
from django.utils.safestring import mark_safe
import markdown
```

Django 对于在模板上显示字符串这个事情是非常谨慎的，比如字符串中包含“>”符号，是显示为大于号还是显示为 HTML 中常见的诸如<p>左边的符号呢？django.utils.safestring 的作用就是将字符串编程为“safe strings”，即实实在在的字符，而不是 HTML 代码，那么这种情况下“>”就是大于号了。mark_safe()方法返回的就是这种“safe strings”。

```
@register.filter(name='markdown')    #①
def markdown_filter(text):    #②
    return mark_safe(markdown.markdown(text))    #③
```

从语句②开始定义选择器函数，其参数 text 就是等待传入的字符串。

语句③返回的是“safe string”，并且是经过 markdown()方法之后返回的结果，使 Django 模板能够顺利接收。

语句①的作用是重命名语句②中的选择器函数，即将名字由 markdown_filter 修改为 markdown。我们在语句②中之所以没有使用 markdown 这个名字，是因为在前面引入的第三方库中有 import markdown，要避免语句②的函数名和引入的第三方库的名字冲突，所以语句②使用了 markdown_filter 的名字，否则语句③中的 markdown.markdown(text)会报错。但是，在模板中使用这个选择器时，还是要按照习惯使用 markdown 这个名字。因此，在函数前面增加语句①，对语句②重命名。

接下来就是在 Django 模板中使用这个选择器了。这里仅演示一个页面，其他部分读者可以自行修改。打开./templates/article/list/article_content.html 文件，找到如下显示文章内容的代码。

```
   <textarea id="append-test" style="display:none;">
{{ article.body }}
   </textarea>
```

将这部分代码用下面的代码替换。

```
{{ article.body | markdown }}
```

完成之后看一下效果。检查 Django 和 Redis 数据库服务是否已经运行，确认运行之后，在浏览器中访问某篇文章，会看到与前面一样的显示效果。这就是使用了 Markdown 选择器的结果。

为了对比，读者还可以把 Markdown 选择器删除，这时查看页面效果，就能理解这个选择器的用途了。

通过本节的实践，Django 越来越让人着迷了，它不仅有很多内置的方法供我们使用，还能自定义某些对象，满足个性化的需要，并且依然保持着快速开发的特点。

4.6.5　知识点

1．表单：CSRF

只要提交表单，就要注意 CSRF 问题，为什么这个问题如此受关注呢？

CSRF（Cross-Site Request Forgery）意为跨站请求伪造，也被称为 one click attack/session riding，缩写为 CSRF/XSRF。

对于 CSRF，可以简单地理解为“有人用你的身份，并以你的名义发送恶意请求”，表现为：以你的名义发送邮件、消息，盗取你的账号，购买商品，虚拟货币转账等。它会造成个人隐私泄露，乃至财产安全等问题。

幸亏，在经过了深刻的教训之后，开发者已经认识到了 CSRF 的危害性，在 Django 中自动对此进行了防范。因此，在提交表单时，必须注意这个问题（如本项目所示）。就 POST 和 GET 两种请求方式而言，GET 毫无安全性，因此只用作数据的查询；而一旦涉及数据的添加、修改、删除，一定要采用 POST 方式。所以，CSRF 也不是多么可怕，只要网站设计符合规范，CSRF 攻击便无从谈起。

2．表单：显示

显示表单可以用 HTML 代码中的<form>和<input>实现，也可以使用{{ form }}变量。

```
>>> class FooForm(forms.Form):
...     name = forms.CharField(max_length=100)
...
>>> print(form)
<tr><th><label for="id_name">Name:</label></th><td><input id="id_name" maxlength="100"
name="name" type="text" required /></td></tr>
```

默认有<label>标签，并且名称和字段一样，只不过首字母大写，后面默认增加一个冒号。

<input>中的 id 也是根据默认原则生成的。在默认状态下，每个字段都不能为空，所以有 required 参数。

```
>>> form2 = FooForm(auto_id="%s_id", label_suffix="-----")
>>> print(form2)
<tr><th><label      for="name_id">Name-----</label></th><td><input      id="name_id"
maxlength="100" name="name" type="text" required /></td></tr>
```

在 form2 这个表单类实例中，使用了参数 auto_id 和 label_suffix，对 HTML 代码中的<label>和<input>中的 id 名称进行了控制。

在模板页面中，可以通过{{ form.as_p }}或者{{ form.as_ul }}的格式显示，默认为{{ form.as_table }}方式。

- {{ form.as_table }}：得到的是用<tr>显示的效果。
- {{ form.as_p }}：得到<p>包裹的 HTML 代码。
- {{ form.as_ul }}：得到按照<li>显示的效果。

在模板页面中，还可以使用渲染对象的各种报错信息。这方面内容官方文档中有详细描述，请读者自行阅读。

其实，刚才建立的 form 也是一个对象，可以按照 Python 中对对象的理解，理解表单的方法和属性。

```
>>> dir(form)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__html__',
'__init__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_bound_fields_cache', '_clean_fields',
'_clean_form', '_errors', '_html_output', '_post_clean', 'add_error',
'add_initial_prefix', 'add_prefix', 'as_p', 'as_table', 'as_ul', 'auto_id',
'base_fields', 'changed_data', 'clean', 'data', 'declared_fields', 'empty_permitted',
'error_class', 'errors', 'field_order', 'fields', 'files', 'full_clean', 'has_changed',
'has_error', 'hidden_fields', 'initial', 'is_bound', 'is_multipart', 'is_valid',
'label_suffix', 'media', 'non_field_errors', 'order_fields', 'prefix',
'use_required_attribute', 'visible_fields']
```

3．文档导读

（1）*Custom template tags and filters*，https://docs.djangoproject.com/en/2.1/howto/custom-template-tags/。

（2）*Working with forms*，https://docs.djangoproject.com/en/2.1/topics/forms/。

4.7 管理和应用文章标签

本节所讨论的是“文章标签”，这不同于前面学习过的“模板标签”。所谓“文章标签”，就是文章的作者可以为自己的文章设置标签（tag），也称为文章的关键词，如图 4-23 所示。

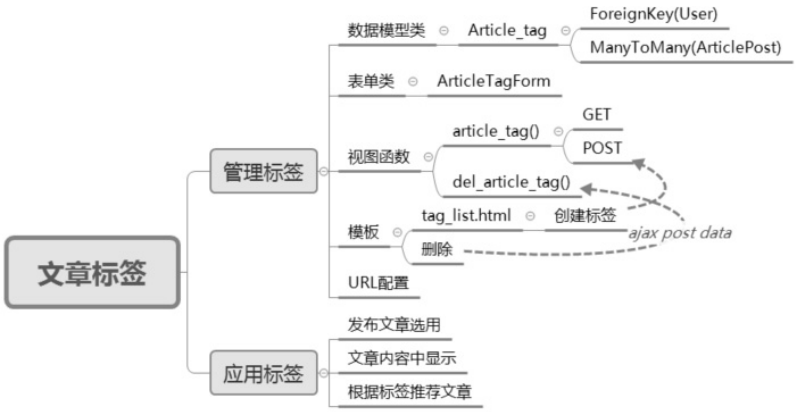

图 4-23　知识技能导图

4.7.1　管理文章标签

有一个名称为 django-taggit 的应用可以直接管理文章标签，如果读者对此有兴趣，请查看其官方网站（https://django-taggit.readthedocs.io/en/latest/index.html），并尝试应用到项目中。

本节我们要做一个属于自己的标签，哪怕这个标签很难看，也是自己动手做的，只有这样才能学习 Django 的功能。所以，本节还是要从事编写数据模型、表单类和视图函数等工作。在开发实践中，笔者鼓励使用类似 django-taggit 这样的第三方应用。

在./article/models.py 中编写一个关于文章标签的数据模型类。

```
class ArticleTag(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE, related_name="tag")
    tag = models.CharField(max_length=500)

    def __str__(self):
        return self.tag
```

这个类要写在 ArticlePost 的前面，然后在 ArticlePost 类里面增加如下代码。

```
article_tag = models.ManyToManyField(ArticleTag,
                                     related_name='article_tag', blank=True)
```

执行迁移数据操作（python manage.py makemigrations 和 python manage.py migrate）。为了满足好奇心，来看一下数据库表结构，如下所示。

```
sqlite> .table
account_userinfo               auth_permission
account_userprofile            auth_user
article_articlecolumn          auth_user_groups
article_articlepost            auth_user_user_permissions
article_articlepost_article_tag  blog_blogarticles
article_articlepost_users_like   django_admin_log
article_articletag             django_content_type
article_comment                django_migrations
auth_group                     django_session
auth_group_permissions

sqlite> pragma table_info(article_articletag);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           tag         varchar(50  1                       0
2           author_id   integer     1                       0
sqlite> pragma table_info(article_articlepost_article_tag);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           articlepos  integer     1                       0
2           articletag  integer     1                       0
```

已经在数据库中看到了两个数据库表 article_articletag 和 article_articlepost_article_tag，分别用来保存文章标签的标签名称和相应的用户 id，以及标签与文章的对应关系。

按照前面的学习步骤，在./article/forms.py 中编写表单类，代码如下。

```
from .models import ArticleTag
class ArticleTagForm(forms.ModelForm):
    class Meta:
        model = ArticleTag
        fields = ('tag', )
```

完成上述步骤后，就可以编写视图函数了。在./article/views.py 中分别将 ArticleTag 和 ArticleTagForm 两个类引入，然后增加如下函数。

```
from .models import ArticleTag
from .forms import ArticleTagForm

@login_required(login_url='/account/login')
@csrf_exempt
def article_tag(request):
    if request.method == "GET":
        article_tags = ArticleTag.objects.filter(author=request.user)
        article_tag_form = ArticleTagForm()
        return render(request, "article/tag/tag_list.html",
                      {"article_tags":article_tags,
                       "article_tag_form":article_tag_form})

    if request.method == "POST":
        tag_post_form = ArticleTagForm(data=request.POST)
        if tag_post_form.is_valid():
            try:
                new_tag = tag_post_form.save(commit=False)
                new_tag.author = request.user
                new_tag.save()
                return HttpResponse("1")
            except:
                return HttpResponse("the data cannot be save.")
        else:
            return HttpResponse("sorry, the form is not valid.")
```

此函数的基本格式与以往类似的函数差不多，也没有增加新的知识，所以不解释了，建议读者对函数中的每一行进行注释，理解其运行过程，遇到疑惑之处，可以参考本书前面的有关说明。

接下来要做的是在./article/urls.py 中增加 URL 设置，代码如下。

```
path('article-tag/', views.article_tag, name="article_tag"),
```

至此，管理文章标签的后端代码就写好了。下面来完成前端代码的编写。

编辑./templates/article/leftslider.html 文件，在有关文章的操作中增加如下一项。

```
<p><a href="{% url 'article:article_tag' %}">文章标签</a></p>
```

确认 Django 和 Redis 服务都启动了，用一个账号登录，然后访问“管理后台”界面（在网站首页单击用户名下拉菜单进入），如图 4-24 所示。

图 4-24　增加了左侧栏项目

当然，现在单击新增加的“文章标签”只能报错，因为我们还没有具体编写针对它的模板。

在./templates/article 目录中创建子目录 tag，并在里面创建 tag_list.html 文件，即./templates/article/tag/tag_list.html，编辑这个文件，输入如下代码。

```
{% extends "article/base.html" %}

{% load staticfiles %}

{% block title %}articles tag{% endblock %}

{% block content %}
<div>    #①
    <p>添加文章标签</p>
     <form class="form-horizontal" action="." method="post">{% csrf_token %}
        <div class="row" style="margin-top: 10px;">
            <div class="col-md-2 text-right"><span>文章标签</span></div>
            <div class="col-md-10 text-left">{{article_tag_form.tag}}</div>
        </div>
        <div class="row">
            <input type="button" class="btn btn-primary btn-lg"
                   style="margin-left:200px; margin-top:10px;"
                   value="添加" onclick="add_tag()">
        </div>
    </form>
</div>
<div>    #②
    <p>已有标签列表</p>
    <table class="table table-hover">
     <tr>
         <td>序号</td>
         <td>文章标签</td>
         <td>操作</td>
     </tr>
     {% for article_tag in article_tags %}
     <tr id={{ forloop.counter }}>
         <td>{{ forloop.counter }}</td>
         <td>{{ article_tag.tag }}</a></td>
         <td>
             <a nane="delete" href="javascript:"
```

```
                onclick="del_tag(this, {{ article_tag.id }})">
                 <span class="glyphicon glyphicon-trash"></span>
               </a>
            </td>
        </tr>
       {% empty %}
        <p>You have no article tags. Please add them.</p>
        {% endfor %}
       </table>
</div>
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
<script type="text/javascript">
    function add_tag(){
        tag = $("#id_tag").val();
        $.ajax({
            url: '{% url "article:article_tag" %}',
            type: "POST",
          data: {"tag":tag},
          success: function(e){
            if(e=="1"){
                layer.msg("You have added a new tag.");
                window.location.reload();
            } else {
                layer.msg(e)
            }
          }
        });
    }
</script>
{% endblock %}
```

这部分代码由两部分组成，语句①下面这部分是一个表单，用于添加新的文章标签；语句②下面这部分用于显示已有的文章标签。

语句①这部分的“添加”按钮使用了 JavaScript 函数 add_tag()，通过 Ajax 方式将新增的文章标签传给后端的视图函数，显示效果如图 4-25 所示。

图 4-25　添加文章标签

已经实现了添加文章标签的功能，还有一个删除文章标签的功能没有实现——单击文章标签列表中的“删除”符号即可实现。

在./article/views.py 中增加一个删除文章标签的视图函数，代码如下。

```
@login_required(login_url='/account/login')
@require_POST
@csrf_exempt
def del_article_tag(request):
    tag_id = request.POST['tag_id']
    try:
        tag = ArticleTag.objects.get(id=tag_id)
        tag.delete()
        return HttpResponse("1")
    except:
        return HttpResponse("2")
```

读者或许会发现，在这个视图函数上方，还有一个删除文章的视图函数，两个视图函数的基本结构是一致的，所以可以考虑写一个统一的删除某对象的方法——如果读者学有余力，可以去研究如何解决。

视图函数编写好后，在./article/urls.py 中增加 URL 的设置，代码如下。

```
path('del-article-tag/', views.del_article_tag, name="del_article_tag"),
```

然后回到./templates/article/tag/tag_list.html 文件，编写单击“删除”图标时所要调用的 JavaScript 函数 del_tag()，相应的代码如下。

```
function del_tag(the, tag_id){
    var article_tag = $(the).parents("tr").children("td").eq(1).text();
    layer.open({
        type: 1,
        skin: "layui-layer-rim",
        area: ["400px", "200px"],
        title: "删除文章标签",
        content: '<div class="text-center" style="margin-top:20px">
                    <p>是否确定删除文章标签《'+article_tag+'》</p>
                </div>',
        btn:['确定', '取消'],
        yes: function(){
            $.ajax({
                url: '{% url "article:del_article_tag" %}',
                type:"POST",
                data: {"tag_id":tag_id},
                success: function(e){
                    if(e=="1"){
                        parent.location.reload();
                        layer.msg("The tag has been deleted.");
                    }else{
                        layer.msg("删除失败");
                    }

                },
            })
        },
    });
}
```

至此，就实现了删除功能。

对文章标签的简单管理功能就这样实现了。这里是对原来已经学习过的知识的复习，子曰“学而时习之，不亦说乎”，“温故而知新，可以为师矣”。

4.7.2 发布文章时选择标签

下面要实现的功能是用户发布文章时，能够从已经有的标签中为该文章选定文章标签。

首先在文章发布的页面中，将用户已经添加的标签显示出来。为此要修改./article/views.py 文件中的视图函数 article_post()。这个函数有两个作用，一是满足 GET 请求方式的页面显示，二是满足 POST 请求方式的数据存储。按照开发顺序，先修改 GET 请求，使得页面显示用户设置的文章标签，并作为发布文章的选项，代码如下。

```
def article_post(request):
    if request.method=="POST":
        #原有代码暂不修改，从略
    else:
        article_post_form = ArticlePostForm()
        article_columns = request.user.article_column.all()
        article_tags = request.user.tag.all()    #①
        return render(request, "article/column/article_post.html",
                      {"article_post_form":article_post_form,
                       "article_columns":article_columns,
                       "article_tags":article_tags})    #②
```

语句①是这个视图函数新增的部分，作用是得到当前用户的所有文章标签。语句②增加了"article_tags":article_tags，将得到的文章标签渲染到模板文件。

编辑./templates/article/column/article_post.html 模板文件，在选择文章所属“栏目”代码块下面增加“文章标签”代码块，代码如下。

```
<div class="row" style="margin-top: 10px;">
   <div class="col-md-2 text-right"><span>文章标签:</span></div>
   <div class="col-md-10 text-left">
      {% for tag in article_tags %}
      <label class="checkbox-inline">
         <input class="tagcheckbox" type="checkbox" id="{{ tag.id }}" name="article_tag"
value="{{ tag.tag }}"> {{ tag.tag }}
      </label>
      {% empty %}
      <p>You have not type tags for articles. Please <a href="{% url
'article:article_tag' %}">input your tags</a></p>
      {% endfor %}
   </div>
</div>
```

这样就实现了“文章标签”的显示，并且以多选项选择的方式，允许用户在发布文章时设置文章的标签，其效果如图 4-26 所示。

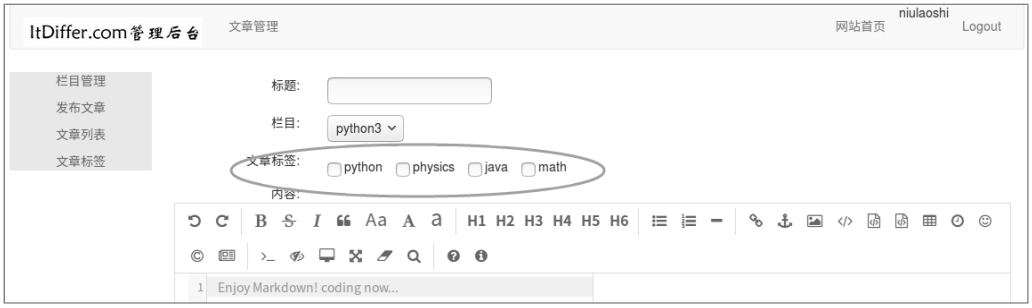

图 4-26　发布文章时选择文章标签

这仅仅实现了展示，当单击“发布”按钮向后台提交文章时，通过 JavaScript 函数 publish_article()得到各项内容，并以 Ajax 的方式发给后端的视图函数。所以，要对该函数进行修改，以得到用户选择的文章标签。继续修改./templates/article/column/article_post.html 文件中的代码，如下所示。

```
<script type="text/javascript" src="{% static 'js/json2.js' %}"></script>    #③
<script type="text/javascript">
    function publish_article(){
        #其他从略
       var article_tags = [];    #④
       $.each($("input[name='article_tag']:checked"),
              function(){article_tags.push($(this).val());});    #⑤
       $.ajax({
         url: "{% url 'article:article_post' %}",
         type: "POST",
         data: {"title":title, "body":body, "column_id":column_id,
"tags":JSON.stringify(article_tags)},    #⑥
         #其他不变，从略
    }
</script>
```

语句③是新引入的一个 JavaScript 插件，可以到 https://github.com/douglascrockford/JSON-js/blob/master/json2.js 下载，其作用在语句⑥中体现了出来，JSON.string()函数就来自于语句③引入的插件，将语句④所定义的 Array 类型的数值转换为 JSON 对象。

语句④和语句⑤是配合使用的，语句④定义了一个数组，用语句⑤得到所选择的项目（选择的文章标签），并将项目加入语句④所定义的数组中。

当数据被提交到后端的视图函数（./article/views.py 中的 article_post()函数）后，视图函数要接收并处理数据，所以相应地还要修改该视图函数（前面修改了响应 GET 请求的部分，现在要修改响应 POST 请求的部分）。

```
import json    #注意，要在文件顶部引入 JSON 模块

def article_post(request):
    if request.method=="POST":
        article_post_form = ArticlePostForm(data=request.POST)
        if article_post_form.is_valid():
            try:
```

```
            #省略原来的部分代码
            new_article.save()
            tags = request.POST['tags']    #⑦
            if tags:
                for atag in json.loads(tags):    #⑧
                    tag = request.user.tag.get(tag=atag)    #⑨
                    new_article.article_tag.add(tag)    #⑩
            return HttpResponse("1")
        #省略其他部分代码
```

语句⑦得到前端所传过来的文章标签，注意是 JSON 格式的，如果要在 Python 中使用，就要用语句⑧中的 json.loads()将 JSON 格式的数据转换为列表。不要忘记，要在此文件的顶部引入 JSON 模块。JSON 模块是 Python 的标准库之一。

语句⑧中的 json.loads(tags)得到了以文章标签为元素的列表，循环此列表，得到每个标签，并用语句⑨得到该文章标签对象。因为在建立文章标签数据模型类之后，又在文章的数据模型类（./article/models.py 中的 ArticlePost 类）中增加了 article_tag = models.ManyToManyField (ArticleTag, related_name='article_tag', blank=True)，即建立了 ArticlePost 和 ArticleTag 之间的多对多关系，所以语句⑩就实现了将文章和文章标签之间对应关系记录的功能。

经过上面的几番修改，就实现了在发布文章时作者可以为文章选定标签的功能。

4.7.3 在文章中显示文章标签

已经为文章添加了标签，那么在显示文章时，也要将标签显示出来。

实现这个功能是比较简单的，在./templates/article/list/article_content.html 文件的作者名称、点赞计数的那一行代码下面增加如下代码即可。

```
<p> <span style="margin-right: 10px"><strong>Tags:</strong></span>
{{ article.article_tag.all | join:", "}}</p>
```

之所以如此简单，主要得益于 Django 模板语言的强大。{{ article.article_tag.all | join:", "}}中的 article.article_tag.all 得到所有的文章标签内容，并且用选择器 join 使结果以“,”连接。

可以查看页面效果（注意检查 Django 服务和 Redis 服务是否运行），如图 4-27 所示。

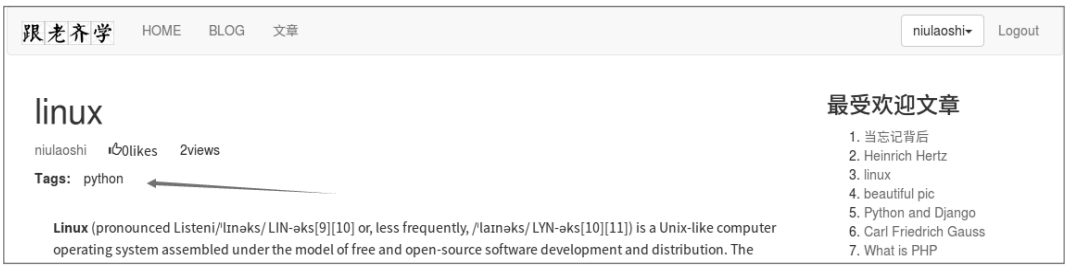

图 4-27　文章中显示标签

再一次展现了 Django 快速开发的特点。

4.7.4　推荐相似文章

在发布文章时，给每篇文章都设置了文章标签，在这个功能的基础上，可以向阅读某篇文章的用户推荐与该文章相似的文章。有同样标签的文章应该是相似的，并且假设用户想阅读相似的文章，于是就有了推荐相似文章的开发需求。

当然，这是一种非常简单的文章推荐功能，目前在大型电子商务网站上做的真正推荐，都是使用机器学习技术实现的。简单也罢，毕竟能够“推荐”，更何况，都要从简单的开始，所以，先完成基于文章标签的简单推荐。为此，要编辑./article/list_views.py 文件中的 article_detail()视图函数。

首先在此文件的顶部完成如下模块引入。

```
from django.db.models import Count
```

然后在./article/list_views.py 的 article_detail()函数的 return 语句前面增加如下语句。

```
article_tags_ids = article.article_tag.values_list("id", flat=True)    #①
similar_articles = ArticlePost.objects.filter(article_tag__in=article_tags_ids).
                                       exclude(id=article.id)    #②
similar_articles = similar_articles.annotate(same_tags=Count("article_tag")).
                                    order_by ('-same_tags', '-created')[:4]    #③
```

语句①中 values_list()的作用就是得到 article 对象的属性 article_tag 的 id 列表。请读者不要忘记，ArticlePost 和 ArticleTag 之间是多对多关系，./article/models.py 中的 ArticlePost 类的属性 article_tag = models.ManyToManyField(ArticleTag, related_name='article_tag', blank=True)声明了这种关系，所以可以通过 article.article_tag 得到其文章标签的 id 值。为了进一步理解 values_list()，可以在交互模式中做如下尝试。

```
>>> from article.models import ArticlePost
>>> article = ArticlePost.objects.get(id=20)
>>> article.article_tag.values_list("id")
<QuerySet [(2,), (3,)]>
```

如果只声明字段值“id”，返回的是由元组组成的列表，也可以声明两个字段，如下所示。

```
>>> article.article_tag.values_list("id", "tag")
<QuerySet [(2, 'python'), (3, 'PHP')]>
```

返回值也是元组组成的列表。但是，如果使用了参数 flat=True，结果就有所变化了。

```
>>> article.article_tag.values_list("id", flat=True)
<QuerySet [2, 3]>
```

在 Django 中，除 values_list()外，还有类似的 values()。它们有相同的地方，即得到当前对象的所有字段或者指定字段的值；同时也有区别，values()返回值的类型是字典，而 values_list 返回值的类型是列表（如果不声明 flat=True，列表是由元组组成的）。详细内容可以阅读官方文档（https://docs.djangoproject.com/en/2.1/ref/models/querysets/#values-list）。

语句①就得到了当前文章的所有标签(article tag)在数据库表中的 id 值，并且以列表的形式返回结果。

语句②可以看成两个部分，第一部分是 ArticlePost.objects.filter(article_tag__in=article_tags_ids)，这是一个条件选择指令，找出文章标签的 id 在 article_tags_ids（列表）里面的所有 ArticlePost 对象（文章）；第二部分是 exclude(id=article.id)，exclude()是一个条件选择函数，其含义是排除参数所规定的值，即从第一部分筛选出来的结果中，将当前文章清除。这样语句②得到了所有与当前文章有共同文章标签的文章对象，这些对象就是相似文章。

语句③使用了本章前面用过的方法，对所有相似文章，根据与当前文章相同的标签数量进行标注，然后以相同标签数量和文章发布时间为关键词排序（倒序）。不要忘记，这里使用了 Count，应该在本文件的顶部引入这个类，即增加 from django.db.models import Count。

最后就是将 return 语句进行适当修改，代码如下。

```
return render(request, "article/list/article_content.html", \
        {"article":article, "total_views":total_views, \
        "most_viewed": most_viewed, "comment_form":comment_form, \
        "similar_articles":similar_articles})
```

视图函数修改好之后，修改相应的模板文件./templates/article/list/article_content.html，找到右边的代码编写位置，在“最多评论文章”栏目的代码块下面增加如下代码。

```
<hr>
<p class="text-center"><h3>推荐相似文章</h3></p>
{% for similar in similar_articles %}
<p><a href="{{ similar.get_url_path }}">{{ similar.title }}</a></p>
{% empty %}
<p>Sorry, 没有相似文章</p>
{% endfor %}
```

请读者检查 Django 和 Redis 服务是否都已经运行起来了，并且建议多发布几篇文章，以便测试刚才创建的功能，如图 4-28 所示。

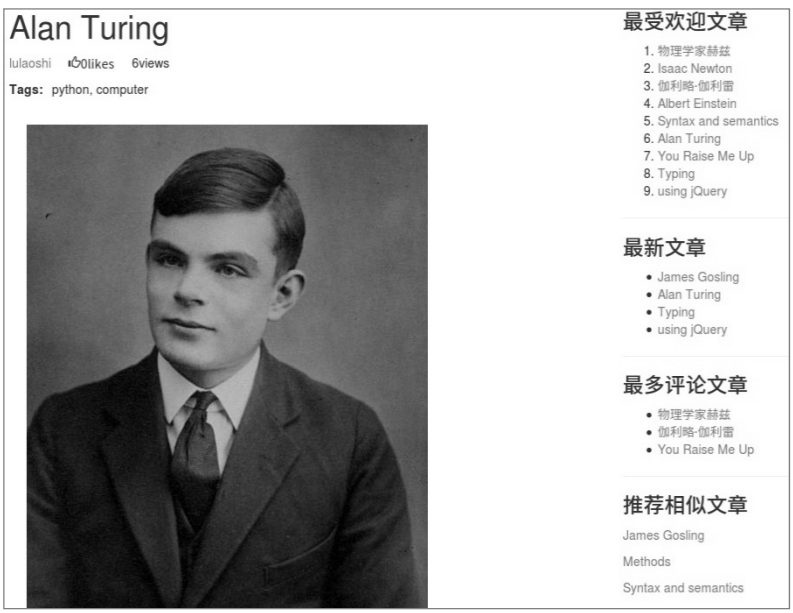

图 4-28　显示相似文章

从“推荐相似文章”栏目中可以浏览与本篇文章有相同标签的其他文章。

我们的 Django 项目发展到现在，已经建立起了一个具有用户管理和文章管理、浏览等功能的多用户系统，并且涉及了诸多 Django 的知识。当然，在 Web 开发的道路上，现在也只能说刚刚开始，后面还要保持旺盛的精力不断进取，最终才能体会到成功的喜悦。

4.7.5　知识点

1．HTTP 404

在网站开发中，404 是最著名的数字了，其全称是“HTTP 404”，是 HTTP 的状态码。当用户提交访问请求时，服务器没有响应且不知原因，就会出现“server not found”提示。代码 404 的第一个“4”代表客户端的错误，如错误的 URL；后两位数字则代表着特定的错误信息。HTTP 的三字符代码与早期通信协议 FTP 和 NNTP 的代码类似。

在网站开发中，不仅有 404，HTTP 的状态码还有不少，以下是一些常见的状态码（节选自维基百科的“HTTP 状态码”词条）。

- 200 OK：请求已成功，请求所希望的响应头或数据体将随此响应返回。
- 302 Found：请求的资源现在临时从不同的 URI 响应请求。这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在 Cache-Control 或 Expires 中进行了指定，这个响应才是可缓存的。新的临时性的 URI 应当在响应的 Location 域中返回。除非这是一个 HEAD 请求，否则响应的实体中应当包含指向新的 URI 的超链接及简短说明。如果这不是一个 GET 或者 HEAD 请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。注意，虽然 RFC 1945 和 RFC 2068 规范不允许客户端在重定向时改变请求的方法，但是很多现存的浏览器将 302 响应视为 303 响应，并且使用 GET 方式访问在 Location 中规定的 URI，而无视原先请求的方法。状态码 303 和 307 被添加了进来，用以明确服务器期待客户端进行何种反应。
- 400 Bad Request：包含语法错误，当前请求无法被服务器理解。除非进行修改，否则客户端不应该重复提交这个请求。
- 403 Forbidden：服务器已经理解请求，但是拒绝执行它。与 401 响应不同的是，身份验证并不能提供任何帮助，而且这个请求也不应该被重复提交。如果这不是一个 HEAD 请求，而且服务器希望能够讲清楚为何请求不能被执行，那么就应该在实体内描述拒绝的原因。当然服务器也可以返回一个 404 响应，假如它不希望让客户端获得任何信息。
- 408 Request Timeout：请求超时。客户端没有在服务器预备等待的时间内完成一个请求的发送。客户端可以随时再次提交这个请求而无须进行任何更改。
- 500 Internal Server Error：服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题会在服务器的程序码出错时出现。

想了解更多的 HTTP 状态码，可以访问维基百科并阅读相应的词条内容。

2．文档导读

（1）*django-taggit*，http://django-taggit.readthedocs.io/en/latest/forms.html。

（2）django-taggit 的中文处理，https://github.com/yangyubo/django-taggit。

第5章

收集和展示图片

图片是网站不可缺少的元素，甚至有人说现在是“读图时代”。因此，本书专门安排一章讲解图片的收集和展示功能。收集就是将网上看到的图片保存下来，展示就是在页面中向浏览者展示自己收集到的图片。

5.1 收集网络图片

网上有很多用户喜欢的图片，人总有一种癖好，自己喜欢的就想纳为己有。现在，数字化的东西可以让我们与他人分享，同时自己也不失去。

本节就完成一个功能，允许用户通过图片地址将图片收藏到自己的账户中，并保存在服务器中。

将本章的应用与前面发表文章的应用区分开来，新建一个应用。进入本项目的根目录中，执行如下命令。

```
$ python3 manage.py startapp image
```

上述代码新建了一个名为 image 的应用。本章的所有开发代码都放在这个应用中。

新建的应用要在项目中声明，即在./mysite/settings.py 文件的 INSTALLED_APPS 的值列表中增加 image 应用，代码如下。

```
INSTALLED_APPS = [
    #其他应用
    'image',
    ]
```

在正式开发之前，还要做一项准备——安装 Pillow，它是 Python 中的一个图片支持模块（详细内容可以访问 http://pillow.readthedocs.io/en/latest/index.html）。

```
$ pip3 install Pillow
```

准备工作已经完成，下面就开始编写实现功能的代码，如图 5-1 所示。

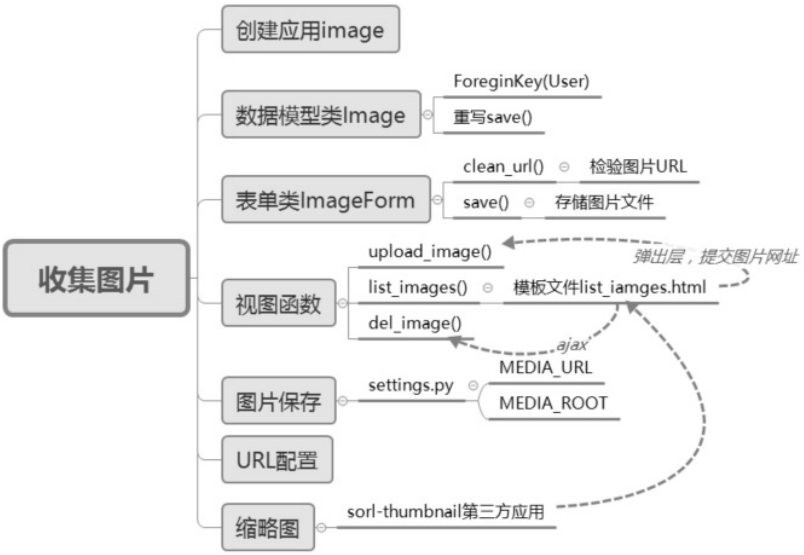

图 5-1　知识技能导图

5.1.1　创建图片相关类

为了能够实现图片的上传和保存，必须要建立数据模型类和表单类。数据模型类用于保存图片的相关信息，对应着数据库表；表单类用于图片上传的数据检验和保存等。

编辑./image/models.py 文件，输入如下代码，创建 Image 类。

```
from django.db import models
from django.contrib.auth.models import User

from slugify import slugify    #①

class Image(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="images")
#②
    title = models.CharField(max_length=300)    #③
    url = models.URLField()    #④
    slug = models.SlugField(max_length=500, blank=True)    #⑤
    description = models.TextField(blank=True)    #⑥
    created = models.DateField(auto_now_add=True, db_index=True)    #⑦
    image = models.ImageField(upload_to='images/%Y/%m/%d')    #⑧

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs):    #⑨
        self.slug = slugify(self.title)
        super(Image, self).save(*args, **kwargs)
```

语句①引入 slugify()，这个方法在本书前面已经使用过了，如果读者一时忘记，可以返回复习。

Image 类和 User 类属于多对一（一对多）的关系，它们通过语句②的外键实现对应关系，从而确定某张图片的拥有者是这个用户。

语句③定义了图片的标题。

语句④专门用来存储网络图片的 URL。这里使用了 URLField 类，这个类继承了 CharField 类。它们的区别在于 URLField 中如果不规定最大长度，则默认是 200，即为存储 URL 做了专门的属性设置。

语句⑤定义了 Image 对象的 slug 字段，与本书前面提到的一样，用在 URL 显示上。

语句⑥用于存储描述图片的文本内容。

语句⑦是 Image 实例的创建时间（保存该图片的时间），auto_now_add=True 表示当这个类的实例被创建时，时间值会自动创建并写入数据库；db_index=True 意味着用数据库的此字段作为索引。

语句⑧规定了图片上传到服务器的物理存储地址。ImageField 类继承了 FileField 类，所以能够接收上传的图片，其中参数 upload_to 规定了所上传的图片文件的存储路径。

语句⑨重写了父类的 save 方法，当通过 Image 实例调用此方法时，自动实现 slug 的生成（slugify(self.title)）和存储。

Image 类写好之后，就要迁移数据库，以生成相应的数据库表。用如下方式查看数据库表结构：

```
sqlite> pragma table_info(image_image);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           title       varchar(30  1                       0
2           url         varchar(20  1                       0
3           slug        varchar(50  1                       0
4           descriptio  text        1                       0
5           created     date        1                       0
6           image       varchar(10  1                       0
7           user_id     integer     1                       0
```

图片的数据模型创建完成后，就要编写表单类了。本项目的功能需求是用户提交图片网址，然后由程序将该图片下载并保存到指定的位置。

创建./image/forms.py 文件，并在其中增加如下代码。

```
from django import forms
from django.core.files.base import ContentFile
from slugify import slugify
from urllib import request

from .models import Image

class ImageForm(forms.ModelForm):
```

```
    class Meta:
        model = Image
        fields = ('title', 'url', 'description')

    def clean_url(self):    #⑩
        url = self.cleaned_data['url']    #⑪
        valid_extensions = ['jpg', 'jpeg', 'png']    #⑫
        extension = url.rsplit('.', 1)[1].lower()    #⑬
        if extension not in valid_extensions:    #⑭
            raise forms.ValidationError("The given Url does not match valid image
extension.")
        return url    #⑮

    def save(self, force_insert=False, force_update=False, commit=True):    #⑯
        image = super(ImageForm, self).save(commit=False)    #⑰
        image_url = self.cleaned_data['url']
        image_name = '{0}.{1}'.format(slugify(image.title),
                                      image_url.rsplit('.', 1)[1].lower())
        response = request.urlopen(image_url)    #⑱
        image.image.save(image_name, ContentFile(response.read()), save=False)    #⑲
        if commit:
            image.save()

        return image
```

对于表单类，其基本结构和作用在本书前面都已经介绍过了，读者并不陌生。不过，在上述代码中，又出现了新内容。语句⑩就是新出现的一个函数，其作用主要是处理某个字段值。这个函数的命名规则是 clean_<fieldname>，其中 fieldname 就是数据模型类中的字段名称，例如语句⑩函数名称 clean_url 中的“url”就是 Image 类中的字段“url”。这个函数除 self 外，不传入其他参数，更不直接得到表单提交的内容，而是通过 self.cleaned_data 获取相应字段的值，即语句⑪所示的方式。

语句⑫规定了图片的扩展名，语句⑬则是从得到的图片的网址中分解出其扩展名，然后在语句⑭中进行判断。如果属于所规定的扩展名，就认为提交的 URL 对象是一个图片，并且符合规定的要求。这种判断对象是否为图片的方法比较简单，因为本书在这里的终极目标不是判别对象是否是图片，所以暂时用这个方法。在真实项目中，建议用专门的方法判断一个对象是否为图片，而不是仅凭其扩展名。

最后，语句⑮将经过验证后的字段值返回。

如读者所知，ModelForm 类中有一个 save()方法，其作用就是将表单提交的数据保存到数据库。ImageForm 类是 ModelForm 类的子类，当建立了 ImageForm 类的实例后，该实例也拥有了 save()方法，即实现实例数据的保存，这是我们在前面已经应用过的。但是，语句⑯重写了 save()这个方法，当通过 ImageForm 类实例调用 save()方法时，调用的是语句⑯。

语句⑰中依然执行父类 ModelForm 的 save()方法，commit=False 意味着实例虽然被建立了，但并没有保存数据。

语句⑱中 request 不是视图函数中的参数 request，而是 Python 标准库 Urllib 中的一部分，是一个很好的爬虫工具。request.urlopen(image_url)的作用是以 GET 方式访问该图片地址，就是模拟用户把网址输入到浏览器的地址栏中，之后在浏览器中返回相应的内容，这里同样返回一个 Request 对象。通过该对象得到所访问 URL 的数据（图片的 ASCII 字符串），在语句⑲中使用 response.read()就是要得到此数据。

语句⑲是将语句⑱中返回的结果保存到本地，并按照约定的名称给该图片文件命名。ContentFile 类是 Django 中 File 类的子类，它接收字符串为参数。这里直接使用了 image 属性或字段（第二个 image）的 save()方法。在 Image 类中有 image = models.ImageField (upload_to='images/ %Y/%m/%d')，属性是 ImageField 类型。ImageField 继承了 FileField，而 FileField 具有一个 save()方法，其官方文档中显示的样式是 FieldFile.save(name, content, save=True)。因此，语句⑲采用这种方式保存图片。

5.1.2 收集和管理图片

要收集网上的图片，就必须提交图片的地址，所以要有视图函数接收所提交的图片 URL 等信息。此外，仅保存了还不够，用户还要浏览和管理，比如删除图片。所以，要在./image/views.py 文件中编写两个视图函数，专门用来处理上述问题，代码如下。

```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from django.views.decorators.csrf import csrf_exempt
from .forms import ImageForm
from .models import Image

@login_required(login_url='/account/login/')
@csrf_exempt
@ require_POST
def upload_image(request):
    form = ImageForm(data=request.POST)    #①
    if form.is_valid():
        try:
            new_item = form.save(commit=False)    #②
            new_item.user = request.user
            new_item.save()    #③
            return JsonResponse({'status':"1"})    #④
        except:
            return JsonResponse({'status':"0"})
```

上述视图函数的作用是处理前端表单提交的图片 URL 及其相关信息。

语句①利用提交的数据建立了表单类的实例。语句②使用该实例的 save()方法，但这里没有保存数据（commit=False）。当执行语句③时，该图片被保存到本地指定目录中。

语句④返回的是 JSON 数据，这与前面返回字符串稍有不同，当然也可以返回字符串，只不过这里给读者展示另外一种返回的数据类型。

upload_image()是接收并处理前端提交的数据的函数，下面还要继续编写展示图片的函数list_images()，代码如下。

```
@login_required(login_url='/account/login/')
def list_images(request):
    images = Image.objects.filter(user=request.user)
    return render(request, 'image/list_images.html', {"images": images})
```

上面两个视图函数编写完之后，为了能够让本应用被项目接收，还要在./mysite/urls.py 文件中增加应用的 URL 配置，代码如下。

```
urlpatterns = [
    #省略其他部分
    path('image/', include('image.urls', namespace='image')),
]
```

然后在本应用中创建./image/urls.py 文件，进行 image 应用相关的 URL 配置，代码如下。

```
from django.urls import path
from . import views
app_name = 'image'
urlpatterns = [
    path('list-images/', views.list_images, name="list_images"),
    path('upload-image/', views.upload_image, name='upload_image'),
]
```

下面做一些零碎的准备工作。

前面对文章管理的部分已经建立了一个后台，此处依然采用类似的方式，用户在后台管理中能够添加、删除图片。

依然使用原来的后台导航和侧边栏功能的模板文件。

将图片列表的功能添加到后台导航模板文件中，即修改./templates/article/header.html 文件，添加如下代码。

```
<li><a href="{% url 'article:article_column' %}">文章管理</a></li>
<li><a href="{% url 'image:list_images' %}">图片管理</a></li>    <!—新增加的部分-->
```

然后修改./templates/article/leftslider.html 文件，将前面已有的功能命名为“文章管理”，现在新增加的部分命名为“图片管理”。以下代码放在原有代码的后面。

```
<hr>
<div class="text-center" style="margin-top: 5px;">
    <p><h4>图片管理</h4></p>
    <p><a href="{% url 'image:list_images' %}">图片管理</a></p>
</div>
```

准备工作完毕，开始创建./templates/image 目录，将本应用的模板文件都放到这个目录中。

在此目录中创建模板文件 list_images.html，其代码如下。

```
{% extends "article/base.html" %}

{% load staticfiles %}
```

```
{% block title %}images{% endblock %}

{% block content %}
<div>
    <button type="button" class="btn btn-primary btn-lg btn-block" onclick="addImage()">
添加图片</button>
    <div style="margin-top:10px;">
    <table class="table table-hover">
        <tr>
            <td>序号</td>
            <td>标题</td>
            <td>图片</td>
            <td>操作</td>
        </tr>
        {% for image in images %}
        <tr>
            <td>{{ forloop.counter }}</td>
            <td>{{ image.title }}</td>
            <td>{{ image.image }}</td>
            <td><a nane="delete" href="javascript:" onclick="del_image(this, {{ image.id }})">
<span class="glyphicon glyphicon-trash" style="margin-left:20px;"></span></a></td>
        </tr>
        {% empty %}
        <p>还没有图片，请点击上面的按钮添加图片</p>
        {% endfor %}
    </table>
    </div>
</div>
{% endblock %}
```

运行 Django 服务，在浏览器中访问 http://localhost: 8000/image/list-images/（用户如果没有登录，请先登录），效果如图 5-2 所示。

图 5-2　显示添加图片的功能

接下来，就应该完成单击“添加图片”按钮实现提交图片地址及其标题和描述的功能。还是使用 JavaScript 的方式完成这个功能，读者在刚刚创建的模板文件（./templates/image/list_images.html）中已经看到了，在<button>中使用了 onclick="addImage()"。继续在这个模板中增加如下代码。

```
<script src="{% static 'js/jquery.js' %}"></script>
```

```
<script src="{% static 'js/layer.js' %}"></script>
<script type="text/javascript">
    function addImage(){
        var index=layer.open({
          type: 1,
          skin: 'layui-layer-demo',
          closeBtn: 0,
          shift: 2,
          shadeClose: true,
          title: "Add Image",
          area: ['600px', '440px'],
          content: "<div style='padding:20px'><p>请新增扩展名是.jpg 或.png 的网上照片地址
</p><form><div class='form-group'><label for='phototitle' class='col-sm-2 control-label'>
标题</label><div class='col-sm-10'><input id='phototitle' type='text' class='form-control'
style='margin-bottom:5px'></div></div><div  class='form-group'><label  for='photourl'
class='col-sm-2 control-label'>地址</label><div class='col-sm-10'><input id='photourl'
style='margin-bottom:5px'    type='text'    class='form-control'></div></div><div
class='form-group'><label for='description' class='col-sm-2 control-label'>描述</label>
<div  class='col-sm-10'><textarea  class='form-control'  style='margin-bottom:5px'
row='2' id='photodescription'></textarea></div></div><div class='form-group'><div
class='col-sm-offset-2  col-sm-10'><input  id='newphoto'  type='button'  class='btn
btn-default' value='Add It'></div></div></form></div>",
          success: function(){
            $("#newphoto").on('click', function(){
              var title = $("#phototitle").val();
              var url = $("#photourl").val();
              var description = $("#photodescription").val();
              var photo = {"title":title, "url":url, "description":description};
              $.ajax({
                url: '{% url "image:upload_image" %}',
                type: "POST",
                data: photo,
                success: function(e){
                  var status = e['status']
                  if(status =="1"){
                    layer.close(index);
                    window.location.reload();
                  } else {
                    layer.msg("图片无法获取，请更换图片");
                  }
                },
              });
            });
          },
        });
    }
</script>
```

在上面的 JavaScript 代码中，再次使用了弹出提示框插件。并且要注意，这些代码要写在{% endblock %}以内。

再次刷新页面，单击“添加图片”按钮，在弹出的提示框中输入有关内容，如图 5-3 所示。

图 5-3　添加图片的弹出提示框效果

在网上找一些图片，通过此方式提交给本应用（注意，图片必须是.jpg 或.png 格式），如图 5-4 所示。

图 5-4　添加图片地址后的列表页

当然，这里在显示上还有一些问题，后面需要继续完善，读者重点关注是否把图片提交并保存了。

查看数据模型类 Image 中的字段 image = models.ImageField(upload_to='images/ %Y/%m/%d') 所设置的物理存储位置，是否已经有了刚才上传的图片，如图 5-5 所示。

图 5-5　网上图片保存到本地

数据库也要检查一下，看看是否按照预设把数据都保存了下来，使用的 SQL 命令如下所示。

```
sqlite> select * from image_image;
```

最终结果实现了预期目标。

5.1.3　完善图片管理功能

图片已经实现了上传，如果有图片不喜欢怎么办？删除！所以，列表中的删除功能是必须有的。利用已经学习过的知识就能够完成此功能，编辑./image/views.py 文件，增加如下代码。

```
@login_required(login_url='/account/lobin/')
@require_POST
@csrf_exempt
def del_image(request):
    image_id = request.POST['image_id']
    try:
        image = Image.objects.get(id=image_id)
        image.delete()
        return JsonResponse({'status':"1"})
    except:
        return JsonResponse({'status':"2"})
```

这个视图函数与前文删除文章的视图函数相差无几。这里再次提醒读者，可以思考并尝试写一个更具有通用性的删除函数（或者类），以实现所有类似的删除功能。

之后不要忘记在./image/urls.py 文件中配置 URL，在列表中增加下列值。

```
path('del-image/', views.del_image, name='del_image'),
```

在./templates/image/list_images.html 文件中编写早已经为删除图标设定的 JavaScript 函数 del_image()，代码如下。

```
function del_image(the, image_id){
  var image_title = $(the).parents("tr").children("td").eq(1).text();
  layer.open({
    type: 1,
    skin: "layui-layer-rim",
    area: ["400px", "200px"],
    title: "删除图片",
    content: '<div class="text-center" style="margin-top:20px">
                <p>是否确定删除《'+image_title+'》</p>
              </div>',
    btn:['确定', '取消'],
    yes: function(){
      $.ajax({
        url: '{% url "image:del_image" %}',
        type:"POST",
        data: {"image_id":image_id},
        success: function(e){
          var status = e['status']
          if(status=="1"){
            parent.location.reload();
            layer.msg("has been deleted.");
          }else{
            layer.msg("删除失败");
```

```
        }

      },
    })
  },
});
}
```

这个函数的写法与前文类似函数的写法一样，这里不再解释。

这样就实现了图片的删除功能（请读者自行测试）。

请读者仔细观察，这种删除并不彻底，因为该图片并没有从磁盘上删除，只是从本项目的数据库中删除了。

虽然完成了删除功能，但还有需要改进之处，就是在页面中不显示图片，仅列出图片的标题和地址。因此，要修改./templates/image/list_images.html 文件的代码，将下面的代码：

```
<td>{{ image.image }}</td>
```

替换为类似以下代码：

```
<td><imge src=”url of image”></td>    #url of image 是占位符
```

然后在./mysite/settings.py 文件中增加如下设置。

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

MEDIA_ROOT 声明了相对项目根目录的文件保存地址。前面我们已经将图片保存到了相对项目根目录的 images 中，即./images。考虑到以后还可能上传视频等媒体文件，所以还是设置一个统一的目录较好，于是这里设置了一个名为 media 的目录，图片将被保存到./media/images 里面。

MEDIA_URL 用于设置 URL 映射中的路径，下面就编辑./mysite/urls.py 文件，引入 settings 和 static，代码如下。

```
from django.conf import settings
from django.conf.urls.static import static
```

在本文件的 urlpatterns 之后增加如下代码。

```
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

这样就为每个上传的静态图片配置了 URL 路径。做好上面的配置之后，再将上面提到的./templates/image/list_images.html 文件中将要替代的部分用下面的代码替代。

```
<td><img src="{{ image.image.url }}" width="100px" height="100px"></td>
```

确保 Django 服务运行，访问 http://localhost:8000/image/list-images/，原来提交的图片肯定不能显示了，因为按照上面的设置，图片的访问地址已经发生了变化，需要重新提交，并且在提交之后，注意观察文件是否被保存到了./media/images 里面。

最终的展示效果如图 5-6 所示。

图 5-6　列表中的图片展示

这样，就实现了对上传图片的可视化管理。

这里展示的图片虽然已经进行了长宽的设置，但仅仅是在显示的时候对图片尺寸进行了更改，实际上仍然要读取原图。说到这里，读者可能也想到了，能不能使用缩略图呢？当然可以！

sorl-thumbnail 就是很好的 Django 缩略图应用，github.com 上的源码地址是 https://github.com/jazzband/sorl-thumbnail。下面简要演示它在本应用的使用方法，读者可以通过阅读官方文档内容，了解更多的使用方式。这里也再次显示了 Django 快速开发的特点，“不需要重复造轮子”。

首先要做的总是安装，使用 pip 可以快速安装这个应用。

```
$ pip3 install sorl-thumbnail
```

然后在./mysite/settings.py 的 INSTALLED_APPS 中注册名为 sorl.thumbnail 的应用。

```
INSTALLED_APPS = [
   #其他应用…
    'sorl.thumbnail',
]
```

然后修改./templates/image/list_images.html 文件。

在文件中显示缩略图的前面引入 thumbnail（通常放在文件的顶部，但不要放在{% extends "article/base.html" %}之前）。

```
{% load thumbnail %}
```

再将原来显示图片的代码替换为下面的代码。

```
{% thumbnail image.image "100x100" crop="center" as im %}
<td><img src="{{im.url}}" width="{{ im.width }}" height="{{ im.height }}"></img></td>
{% endthumbnail %}
```

重新启动 Django 服务，然后刷新页面，这时候查看到的图片就是缩略图了，如图 5-7 所示。

图 5-7　缩略图效果

图 5-7 所示的缩略图被保存在了./media/cache 目录中。

读者还可以继续创建一个分页功能，因为这个功能和前面文章管理中的分页功能一致，所以笔者就省略了，特别建议读者能够补上。

本节中我们收集了网络图片，并对保存到项目本地的图片进行了管理，还使用了缩略图。因为前面已经奠定了开发基础，所以本节中并没有遇到难点，但并不意味着开发过程会一帆风顺，如果在某些细节上不注意也会有 Bug 出现。随着功能越来越多，重复代码也日渐增加，所以读者要充分使用在 Python 中已经学会的知识，将重复的功能进行抽象，写成能够被公用的类（或函数），虽然笔者在本书中没有这样做。

5.1.4　知识点

1．模型：ImageField

ImageField 专门用来声明某字段是图片类型，其类的完全调用形式如下。

```
class ImageField(upload_to=None, height_field=None, width_field=None, max_length=100,
**options)
```

在字段被设置为 ImageField 类型之后，提交图片，Django 会自动检测所提交的是否是图片类型，不需要用户写检验方法。

在创建 ImageField 实例时，可以指明 upload_to 参数，说明图片保存在服务器上的位置，通常还在 settings.py 中配置 MEDIA_ROOT 和 MEDIA_URL 的值，例如在本书的项目中，代码如下。

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
MEDIA_URL = '/media/'
```

其实，ImageField 继承了 FileField，如果理解了 ImageField，后面再遇到 FileField 也好解决了。阅读下面这段代码。

```
>>> from image.models import Image
>>> image = Image.objects.all()
>>> image
```

```
<QuerySet [<Image: 发放防雾霾口罩>, <Image: 高楼看雾霾>, <Image: 戴口罩晨练>, <Image: Truing>, <Image: Hertz>, <Image: cls>, <Image: zhiling>]>
>>> hertz = image[4]    #选一个记录
>>> hertz
<Image: Hertz>
>>> hertz.user    #该记录的用户名
<User: lulaoshi>
>>> hertz.image    #该记录中的图片
<ImageFieldFile: images/2017/01/24/Hertz.jpg>
>>> hertz.image.name    #图片文件名称
'images/2017/01/24/Hertz.jpg'
>>> hertz.image.path    #图片文件保存的物理地址
'/home/qiwsir/DjangoPracticeProject/mysite/media/images/2017/01/24/Hertz.jpg'
>>> hertz.image.url    #图片的 URL（如果要在浏览器中访问，前面加上主域名和端口）
'/media/images/2017/01/24/Hertz.jpg'
```

当然，对这个文件也可以继续按照文件对象的方式进行操作，比如：

```
>>> import os
>>> from django.conf import settings
>>> old_path = hertz.image.path
>>> hertz.image.name = "images/2017/01/24/Hertz_new.jpg"
>>> new_path = settings.MEDIA_ROOT + hertz.image.name
>>> os.rename(old_path, new_path)
>>> hertz.save()
```

只有在执行了实例的 save()方法后，保存在服务器物理空间中的文件名才会改变，否则只是数据库表记录中的文件名被修改了，物理空间中的文件名并没有变化，在调用时会找不到对应的文件。

不要忘记，如果要上传图片，还要依赖一个名为 Pillow 的库。

2．文档导读

（1）*ImageField source*，https://docs.djangoproject.com/en/2.1/ref/models/fields/#imagefield。

（2）*Thumbnails for Django*，https://github.com/mariocesar/sorl-thumbnail。

（3）*Discussion about Thumbnail Contribution for Django*，https://code.djangoproject.com/wiki/ThumbNails。

5.2　展示图片

已经完成了用户收集图片的功能，下面就是展示图片，允许用户浏览并查看图片的详细内容，如图 5-8 所示。

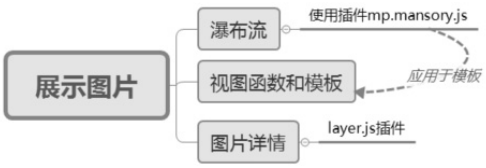

图 5-8　知识技能导图

5.2.1 瀑布流方式展示图片

“瀑布流”是一种网站页面布局方式，在视觉上表现为参差不齐的多栏布局，随着页面滚动条向下滚动，这种布局还会不断加载数据块并附加至当前尾部。最早采用此布局的网站是 Pinterest，之后逐渐在我国流行起来。

当然，网页展示是多变的，在这里笔者使用了瀑布流的方式展示图片，并不意味着在日后还流行这种用法，而且“瀑布流”本身的布局样式也有变化，比如图片是参差不齐的还是整齐排列的等。这里仅仅是一个样例罢了。

此处我们使用来自 https://github.com/jmlp-131092/mp-mansory.js 的插件，实现以瀑布流方式展示用户所收集的各种图片。随着需求和技术的变化，已经发展出更多的瀑布流布局方式，读者如果不喜欢此处选择的插件，可以使用其他插件，使用方法都是类似的。

首先下载上述插件，将其中的 mp.mansory.min.js 文件复制到./static/js 目录中，然后在./static/css 目录中创建 mansory-style.css 文件，并输入如下代码（根据原插件的 style.css 代码进行了适当修改）。

```
#my-gallery-container {
    background-color: #ededed;
}

.falls_item {
    border: #c6c6c6 2px solid;
    width: 100%;
    background-color: white;
    min-height: 100px;
    padding: 5px;
    margin: 10px 0px;
    box-shadow: rgba(0, 0, 0, 0.5) 0px 2px 2px;
    border-radius: 4px;,
}

.falls_item:hover {
    opacity: 0.5;
}

.falls_item.h150{
    width: 100%;
    min-height: 150px;
}

.falls_item.h200{
    width: 100%;
    min-height: 200px;
}

.falls_item.h250{
    width: 100%;
    min-height: 200px;
}
```

```
.falls_padding{
    padding: 10px 5px;
}

.falls_item img {
   max-width: 100%;
}
```

之所以用上述代码而不使用原插件中的 style.css 文件，是因为原文件中的样式表名称与本项目中已经使用的 bootstrap.css 有重复，如果使用就会覆盖原有名称。为此，在这里重写了该文件，并重新命名。读者可以根据自己的实际情况酌情处理。

首先实现瀑布流方式的展示，也是从编写视图函数开始的。

在./image/views.py 文件中增加如下视图函数的代码。

```
def falls_images(request):
    images = Image.objects.all()
    return render(request, 'image/falls_images.html', {"images": images})
```

然后在./image/urls.py 文件中增加一个 URL 配置，代码如下。

```
path('images/', views.falls_images, name="falls_images"),
```

在本项目的导航栏中已经有“文章”栏目，现在增加“美图”栏目。编辑./templates/header.html 文件中导航部分的代码。

```
<ul class="nav navbar-nav" role="navigation">
   <li><a href="{% url 'article_list:list_articles' %}">文章</a></li>
   <li><a href="{% url 'image:falls_images' %}">美图</a></li>     #①
</ul>
```

语句①是新增加的展示图片的功能。单击“美图”即可看到以瀑布流方式展示的所有用户收集的图片。

在./templates/image 目录中创建 falls_images.html 文件，这就是瀑布流方式展示图片的界面，其代码如下。

```
{% extends "base.html" %}
{% load staticfiles %}
{% block title %}Images{% endblock %}
{% block content %}

<div class="container">
<link rel="stylesheet" href="{% static 'css/mansory-style.css' %}" type="text/css" />
<div id="my-gallery-container">
  {% for image in images %}
    <div class="falls_item h200" data-order="{{image.id}}">
       <img src="{{ image.image.url }}">
       <p>{{ image.title }}</p>
    </div>
  {% endfor %}
</div>
</div>
<script src='{% static "js/jquery.js" %}'></script>
<script src="{% static 'js/mp.mansory.min.js' %}"></script>
```

```
<script type="text/javascript">
    jQuery(document).ready(function ( $ ) {
        $("#my-gallery-container").mpmansory(
            {
                childrenClass: 'falls_item', // default is a div
                columnClasses: 'falls_padding', //add classes to items
                breakpoints:{
                    lg: 3,
                    md: 4,
                    sm: 6,
                    xs: 12
                },
                distributeBy: { order: false, height: false, attr: 'data-order', attrOrder:
'desc' }, //default distribute by order, options => order: true/false, height: true/false,
attr => 'data-order', attrOrder=> 'asc'/'desc'
            }
        );
    });
</script>
{% endblock %}
```

这个界面的写法参考了 https://github.com/jmlp-131092/mp-mansory.js 提供的样例，读者可以根据自己对此插件的理解，编写所需的瀑布流样式。

完成上述代码之后，检查 Django 服务是否已经启动，访问 http://localhost:8000/image/ images/，如果已经收集了若干张图片，就可以看到类似如图 5-9 所示的效果。

图 5-9　瀑布流效果

这就是瀑布流方式展示的图片。

再进一步，还可以查看每张图片的详细信息。

5.2.2　查看图片的详细信息

前面以瀑布流方式列出了所有图片，如果要查看每张图片的详细信息，一种方法是用本书前文中所讲述的类似查看文章详细内容的方法，这种方法此处不再重复，请读者自己完成。下面我们使用另外一种纯粹前端的方法来查看图片的完整信息。

在./templates/image/falls_images.html 文件中找到下面这段代码。

```
<div class="falls_item h200" data-order="{{image.id}}">
    <img src="{{ image.image.url }}">    #①
    <p>{{ image.title }}</p>
</div>
```

用下面的代码替换语句①。

```
<a href="javascript:void(0)"
    onclick="displayImage('{{image.user}}', '{{image.title}}',
                          '{{image.image.url}}', '{{image.description}}',
                          '{{request.get_host}}')">
    <img src="{{ image.image.url }}">
</a>
```

这里主要是在<a>标签中增加了一个 onclick 方法，即 displayImage()函数，并且向这个函数传递了一些值，请读者注意传值的方式，没有直接使用诸如{{image.user}}的方式，而是使用了'{{image.user}}'的方式（外层加一对单引号），其目的在于向 dispalyImage()传递字符串。

然后在本文件中编写 JavaScritp 函数 displayImage()。

因为要实现单击某图片后在弹出提示框展示该图片详细信息的功能，所以再次使用熟悉的弹出提示框插件，代码如下。

```
<script type="text/javascript" src="{% static 'js/layer.js'%}"></script>
```

下面是 dispalyImage()的代码，也放在本文件的<script></script>内。

```
function displayImage(user, title, url, description, host){
    layer.open({
      type: 1,
      title: title,
      skin: 'layui-layer-rim', //加上边框
      area: ['800px', '800px'], //宽高
      content: '<div class="text-center"><img src="http://'+host+url+'"><p>发布者：
'+user+'</p></div><div style="margin-left:10px;">'+description+'</div>',
    });
}
```

检查 Django 是否运行了，然后访问 http://localhost:8000/image/images/，单击某图片，即可查看该图片的详细信息，如图 5-10 所示。

要想在图片展示中有非常友好的操作界面，就必须使用 JavaScript 组件或者框架。就目前流行的各种 JavaScript 组件和框架而言，可谓让人眼花缭乱，读者可以根据自己的喜好和擅长选择使用。

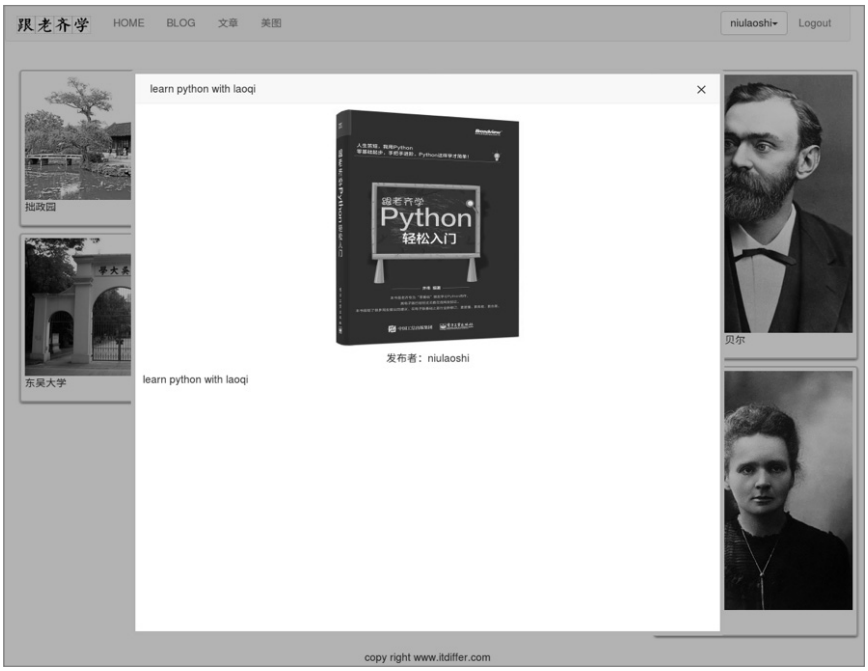

图 5-10　图片详情弹出提示框

虽然我们处在所谓的“读图年代”，甚至是“读视频”，但要想成为这个时代中的佼佼者，还必须读书。笔者借用宋真宗赵恒写下的流传千古的《励学篇》献给读者。

富家不用买良田，书中自有千钟粟。

安居不用架高堂，书中自有黄金屋。

出门莫恨无人随，书中车马多如簇。

娶妻莫恨无良媒，书中自有颜如玉。

男儿欲遂平生志，六经勤向窗前读。

5.2.3　知识点

1．关于 JavaScript

做 Web 开发，如果不使用 JavaScript，一些很酷的效果就难以实现了。首先要非常明确地说明，JavaScript 和 Java 在名字上有点像，在语法上也有相似的地方，但也仅仅是“看上去像”，不要认为 JavaScript 是从 Java 演化来的。事实上，JavaScript 语言在设计方面主要受到了 Self 和 Scheme 两种语言的影响。

JavaScript 诞生于网景公司，虽然现在很多人已经不知道历史上的这家公司了，但是 JavaScript 广为传播，所以使用它的时候请在心里默默感谢网景公司和发明者布兰登 • 艾克（当时就职于网景公司）。

JavaScript 被大量用在 Web 开发中，它不需要进行编译，是一种解释性的脚本语言，可以实现与 HTML 的交互，并且直接嵌入 HTML 页面中，可以单独成为一个单元（比如单独的.js

文件），且实现了结构和行为的分离。一般情况下，可以用它完成如下工作（源自维基百科）。

- 在 HTML 页面加入动态文本。
- 浏览器事件做出响应。
- 写 HTML 元素。
- 数据被提交到服务器之前验证数据。
- 检测访客的浏览器信息。
- 制作 cookies，包括创建和修改等。

为了使开发更加便利，聪明的程序员们制作了各种“轮子”，并且开源。因此，对于使用者来说，不仅要了解 JavaScript，还要知道各种“轮子”，jQuery 就是其中一个。

jQuery 是一个 JavaScript 库，它大大地简化了 JavaScript 编程，故用户很容易学会。jQuery 官方网站是 http://jquery.com/。

除 jQuery 外，还有很多库，可谓多如牛毛，读者要实现某个功能，最好先到网上搜索一下，看看是否有“轮子”可以拿过来使用。这几年有一个名为 React 的库（源码在 https://github.com/facebook/react），宣称“简单、组件化、学一次各处可用”。不仅如此，它还衍生出了 React Native，可用来编写手机上的 App 代码。

在浏览器端，JavaScript“带领”它的众多库，横扫天下，但它并没有满足，凭借 Node.js 进军了服务器端，真的是什么也挡不住了。

Node.js 就是运行在服务器端的 JavaScript，是 Chrome JavaScript 运行时建立的一个平台，还是一个事件驱动 I/O 服务器端 JavaScript 环境，基于 Google 的 V8 引擎。V8 引擎执行 JavaScript 的速度非常快，性能非常好。运行 Node.js 的服务器能支持数万个并发连接。读者如果觉得 Node.js 也值得学，可以参考官网 https://nodejs.org/en/。

千里之行始于足下，不管学习哪个 JavaScript 框架，都要从 JavaScript 入手。

2．文档导读

（1）*JavaScript*，https://docs.djangoproject.com/en/2.1/internals/contributing/writing-code/javascript。

（2）*django-jquery*，https://pypi.python.org/pypi/django-jquery。

（3）*Serializing Django objects*，https://docs.djangoproject.com/en/2.1/topics/serialization。

第6章

中场休整

不是“中场休息”，而是“中场休整”。

进行到这里，如果读者阅读本书并且将代码都敲过，那么就已经达到了 Django 开发的入门级水平。前面几章所讲解的技能，对于 Django 来讲仅仅是一小部分。读者若能够基本理解 Django 的常规工作方式，通常的项目就能够上手开发了，并且每节都有“知识点”和“文档导读”，如果读者对这些内容也认真阅读了，而且对照项目反复实践，那么此时已经进阶到较高水平了。

本章名为“中场休整”，“休整”即“休息整顿”。在前面的项目中，如果读者是“码不停手”地一路很认真地敲过来，当然不会有什么太多的困难，也不会有更多的感悟，所以要“休整”，就是要在原有基础上让自己有所感悟。

针对本章的“休整”，请读者做如下工作。

（1）找一个安静的房间，闭上眼睛，把前面学习的功能和使用到的技能尽可能回忆一番。

（2）请读者将前面随着本书已经辛辛苦苦做好的项目全部删除，一点痕迹也不要留下，留下的都是后患。

（3）一切都从头再来。完全独立地完成前几章中所创建的项目和应用，遇到问题时，使用搜索引擎来解决。记住，这是在创建你自己的项目，不必亦步亦趋地按照本书步骤做，所以你的想法是最重要的。

在读者进行项目开发的过程中，Django 的官方文档是重要的参考资料。此外，许多问题或许已经在 stackoverflow.com 中被解决了，还有本书所提供的 QQ 群也可以帮助读者。

完成上述要求肯定有困难，俗话说“困难只有一个，解决困难的方法有多个”，每克服一个困难，就是编程能力的一次提升。

若读者想做一名真正的程序员，那么就请按照第（1）~（3）条执行。

按照上述步骤进行的中场休整，可能不是 10 分钟或者 15 分钟，不管多长时间，中场休整之后再进入“下半场”，编程思路不仅会豁然开朗，而且技能会得到飞速提升。

下半场，精彩等着你。

第 7 章

创建在线学习应用

本章继续本着“挂一漏万、以点带面、以偏概全”的宗旨，以“在线学习”的应用为载体，深化对 Django 的理解，继续讲解一些新的技能。下面先做好准备工作。

（1）创建一个新的应用。

```
$ python3 manage.py startapp course
```

（2）在./mysite/settings.py 中配置应用。

```
INSTALLED_APPS = [
    #...省略其他
    'course',
]
```

做好了开发准备，就从一个新技能——“基于类的视图”开始。

7.1 基于类的视图

在前述各章中，已经反复编写了视图函数——在视图文件（多数是 views.py）中编写的函数，这种方式被称为“基于函数的视图”，以这种方式能够完成所有 Django 开发中的功能。尽管如此，Django 还提供了一种完成同样功能的代码编写方式——Class-based views，通常称为“基于类的视图”。

暂且不讨论“基于类的视图”与前面所编写的视图函数有什么区别，也不解释为什么还有这样一种写法——“凡是存在的就是合理的，凡是合理的就是存在的”（黑格尔）。我们还是奉行“在做中学”的观点，请读者一边阅读，一遍动手敲代码并进行对比和思考。基于类的视图如图 7-1 所示。

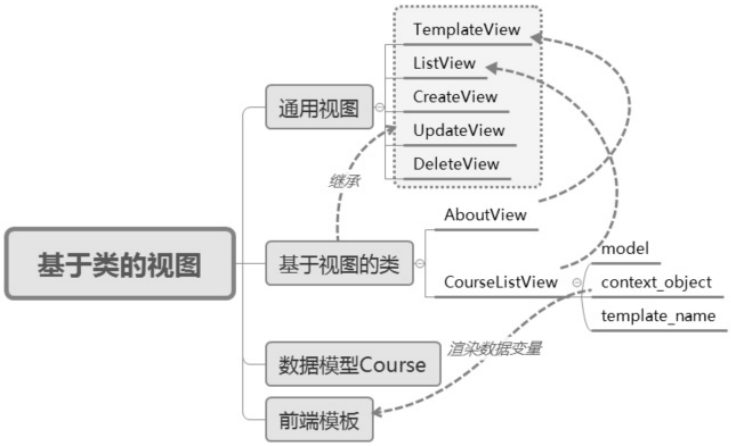

图 7-1　知识技能导图

7.1.1　最简类视图

假设需要在网站中设计一个页面，这个页面用于向浏览者介绍网站，也就是所谓的“关于本站”。请读者运用前面已经学习的技能，想想这个功能应该如何实现？在此驻足几分钟，有了想法，再继续阅读。

读者肯定能够用以往学习的技能实现上述功能，但此处我们换一种方式（这种方法曾经在前文出现过）。

在./mysite/urls.py 中为 course 应用配置 URL，代码如下。

```
path('course/', include('course.urls', namespace='course')),
```

创建./course/urls.py 文件，并输入如下代码。

```
from django.urls import path
from django.views.generic import TemplateView    #①

app_name = "course"
urlpatterns = [
    path('about/', TemplateView.as_view(template_name="course/about.html")),    #②
]
```

这个 URL 的配置与以往有所不同，能发现区别吗？仔细观察，不难发现区别。暂不解释，继续敲代码。

在./templates 这个模板目录中创建一个子目录 course，然后在里面创建文件 about.html，即./templates/course/about.html，输入如下代码。

```
{% extends "base.html" %}
{% block title %}About itdiffer.com{% endblock %}
{% block content %}
<div class="container">
  <h1 class="text-center">Welcome itdiffer.com</h1>
<p class="text-center">www.itdiffer.com 是老齐用来发布自己所编写的教程网站，供广大编程者学习。</p>
```

```
<p class="text-center">QQ 群号：26913719</p>
<h2 class="text-center">欢迎访问。</h2>
</div>
{% endblock %}
```

运行 Django 服务，在浏览器的地址栏中输入 http://localhost:8000/course/about/，效果如图 7-2 所示。

图 7-2　静态网页

成就感油然而生。

仔细思考，是不是感觉上面的过程比以往少了点什么？没有写视图函数，但也能完成访问，其根源就在于语句①和语句②。

语句①引入的 TemplateView 就是最常使用的基于类的通用视图之一，它的作用是返回模板，在语句②中直接使用 TemplateView.as_view()是它的用法之一，并且通过参数 template_name 指定模板。

请重点关注一下语句②，这里没有直接使用 TemplateView(template_name="course/about.html")的方式，而是使用了 TemplateView 的方法 as_view()，这是因为在 URL 解析中，Django 依然只认函数，这与前面在这里写视图函数是类似的。不仅是 TemplateView，后面遇到的其他类也都有 as_view()方法，它作为类的可调用入口，传递给 as_view() 的参数覆盖类中的属性，在语句②中就借此声明了模板文件。

在视图函数中，要向函数传递一个 HttpRequest 对象，需要 request 参数，这是我们已经了解的。在语句②中，也是通过 as_view()函数传给了 request，并且创建类视图的实例，然后调用类视图的 dispatch()方法，注意这个方法并没有在上面显示出来。dispatch()方法会根据不同的访问方式做出响应。

对于刚才的功能，还可以换另外一种写法。

在./course/views.py 中，编写基于类的视图，代码如下。

```
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "course/about.html"
```

除引入 TemplateView 外，再创建一个类 AboutView，它继承 TemplateView，并且设置一个属性 template_name，用来声明模板文件，这与前面语句②中的效果是一样的。那么，在./course/urls.py 中就需要进行修改了（或者将原来的 URL 设置语句注释掉）。

```
from django.urls import path
from .views import AboutView    #③

urlpatterns = [
    path('about/', AboutView.as_view(), name="about"),    ④
]
```

语句③引入了在./course/views.py 中写的类 AboutView，因为它继承了 TemplateView，所以也有 as_view()方法，在语句④中依然使用了这个方法，但是不需要传入 template_name 参数，因为已经在 AboutView 类中声明了。

确保 Django 服务已经启动，再次访问 http://localhost:8000/course/about/，效果同前。

最后，把刚才的“关于本站”放到顶部导航中。打开./templates/header.html 文件，在导航栏中增加如下代码。

```
<li><a href="{% url 'course:about'%}">关于本站</a></li>
```

注意代码的位置。

以上仅对基于类的视图进行了初步讲解，读者可能还没有体会到它的魅力。

7.1.2 读取数据

前面做的都是静态页面，没有与数据库打交道。如果要对数据库进行读、写，该怎么办？

Django 中定义了一系列的通用视图，存放在 django.views.generic 中，包括 ListView、CreateView、UpdateView、DeleteView 等，本节将逐一用到。

为了实现对数据库的操作，先要有数据库表，对此我们已经很熟悉了，需要创建数据模型类。在./course/models.py 文件中创建 Course 类，用来保存课程名称和相关描述，代码如下。

```
from django.db import models
from django.contrib.auth.models import User

from slugify import slugify

class Course(models.Model):
    user = models.ForeignKey(User,  on_delete=models.CASCADE,
                             related_name='courses_user')
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)
    overview = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('-created',)

    def save(self, *args, **kargs):
        self.slug = slugify(self.title)
        super(Course, self).save(*args, **kargs)

    def __str__(self):
        return self.title
```

Course 数据模型建立好之后，执行迁移数据的命令，创建名为 course_course 的数据库表，如下所示，查看数据库表的结构。

```
sqlite> pragma table_info(course_course);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1           1
1           title       varchar(20  1           0
2           slug        varchar(20  1           0
3           overview    text        1           0
4           created     datetime    1           0
5           user_id     integer     1           0
```

为了后续开发方便，先在这个数据库表中插入一条记录，用下面的 SQL 语句完成。

```
sqlite> insert into course_course (title, slug, overview, created, user_id) values
("django web", "django-web", "django is good frame", "2018-06-18", 8);
sqlite> select * from course_course;
id          title       slug        overview            created     user_id
----------  ----------  ----------  ------------------  ----------  ----------
1           django web  django-web  django is good frame  2018-06-18  8
```

然后，使用 ListView 来读取数据库表中课程数据记录。

在./course/views.py 中编写类视图，代码如下。

```
from django.views.generic import TemplateView, ListView    #①

from .models import Course    #②

class CourseListView(ListView):    #③
    model = Course    #④
    context_object_name = "courses"    #⑤
    template_name = 'course/course_list.html'    #⑥
```

语句①引入 ListView，语句②将曾经创建的数据模型类 Course 引入。

重点看语句③及其后的内容。语句③声明了类，并且继承 ListView。语句④声明本类所用到的数据模型，通过语句④就能够得到相应数据库表（此处是 course_course 表）中的所有记录，但是没有使用 Course.objects.all()，而是用语句④的方式非常简洁地表达了。

语句⑤声明了传入模板中的变量名称。如果不写语句⑤，则模板默认变量名称是 object。

语句⑥是已经熟悉的声明模板文件。紧接着就根据这里的路径创建这个模板文件（./templates/course/course_list.html），并且输入如下代码。

```
{% extends "base.html" %}
{% block title %}Course List{% endblock %}
{% block content %}
<div class="container">
  <table class="table table-hover">
    <tr>
      <td>序号</td>
      <td>课程标题</td>
```

```
      <td>讲师</td>
      <td>发布日期</td>
    </tr>
    {% for course in courses %}
    <tr id={{ forloop.counter }}>
      <td>{{ forloop.counter }}</td>
      <td>{{ course.title }}</td>
      <td>{{ course.user.username }}</td>
      <td>{{ course.created }}</td>
    </tr>
    {% endfor %}
  </table>
</div>
{% endblock %}
```

这个模板文件与以往的模板文件没有区别，不再解释。

为了能够浏览，还要在./course/urls.py 文件中增加如下设置（在本文件中首先要引入 CourseListView，即 from .views imort CourseListView）。

```
path('course-list/', CourseListView.as_view(), name="course_list"),
```

启动 Django 服务，访问 http://localhost:8000/course/course-list/，效果如图 7-3 所示。

图 7-3　显示课程列表页

看到了课程列表，把这个功能也放到导航栏中。在./templates/header.html 文件中增加如下代码。

```
<li><a href="{% url 'course:course_list' %}">课程</a></li>
```

再刷新页面，欣赏自己的成果。

在 CourseListView 类的代码中，语句④相当于实现了 Course.objects.all()功能，即将所有记录都读出来。而在某些时候，需要根据条件对数据进行筛选，例如在刚才显示的页面中，只显示某一个讲师的课程，其中一种方法是重写语句④，用下面的语句替代语句④（为了使用下面的语句，不要忘记在文件中引入 User，即 from django.contrib.auth.models import User）。

```
queryset = Course.objects.filter(user=User.objects.filter(username="niulaoshi"))
```

此外，还有一种方法就是重写 get_queryset()方法。这种方法在 Django 的很多类中都有，它的作用就是读取数据库并返回结果（QuerySet）。在 CourseListView 类中增加如下代码。

```
def get_queryset(self):
    qs = super(CourseListView, self).get_queryset()     #⑦
    return qs.filter(user=User.objects.filter(username="niulaoshi"))    #⑧
```

语句⑦调用了父类的 get_queryset()方法，语句⑧则对得到的对象依据条件进行筛选。

以上对基于类的视图做了初步讲解，这仅仅是开始。

7.1.3　初步了解 Mixin

CourseListView 类继承 ListView 类，CourseListView 是单继承。在 Python 的类中，除能够单继承外，还能够实现多重继承，即一个类可以有多个父类（对此有疑惑的读者，请阅读《跟老齐学 Python：轻松入门》中的有关章节）。Django 中有一种常用的继承方式——Mixin，可用于多重继承，一些人把它翻译为“迷信”。什么是 Mixin？

Mix-in（注意写法）本不是编程术语，它是指一种冰激凌和另外一种风味的甜食混合而成的食物，Mix 即英文 mixture 的缩写。后来，在面向对象的编程语言中出现了 Mixin，这个术语是否还有“混合”的含义呢？

在类 CourseListView 中，我们要设定 model=Course，这时声明在当前类中读取数据的对象就是 Course，随后在增加、删除等类中也都要有 model=Course，同样的代码要不断重复写。编码要尽可能避免重复，这是程序员的共识，避免重复的一种有效方法是使用“类继承”。我们可以先写一个类，比如这个类就是包含 model=Course 的，后面用到它时就继承这个类，而不用重复写这句话。

写一个类供其他类继承，的确是一个好方法。为了更明显地标记这个类，使其在名称上就可以体现出它的存在价值就是被别的类继承，或者类比 Mix-in 那种食物，这个类就是要与别的类混合在一起用的，所以在命名时常常以 Mixin 结尾。

在./course/view.py 中输入如下代码。

```
class UserMixin:    #①
    def get_queryset(self):
        qs = super(UserMixin, self).get_queryset()
        return qs.filter(user=self.request.user)

class UserCourseMixin(UserMixin):    #②
    model = Course

class ManageCourseListView(UserCourseMixin, ListView):    #③
    context_object_name = "courses"
    template_name = 'course/manage/manage_course_list.html'
```

以上代码用于用户登录后，进入“后台管理”，对课程进行“增删改查”等操作。

语句①创建了类 UserMixin，表示这个类将被用于后面的类中，而不是作为视图使用。

语句②还是一个 Mixin，但它继承了 UserMixin，意味着语句①中所定义的方法也被带入到语句②中。

请读者认真观察语句③中的继承顺序，这种顺序是有讲究的，一般将 Mixin 类放在左边，其他类放在右边。这里显然是多重继承，类 UserCourseMixin 所代入的就是在语句①和语句②两个类中所定义的方法和属性。类似地，还能够创建其他的类，继承语句①或者语句②中的类，从而不必在类中重复相同的代码。

这就是 Mixin。

在 Django 中，已经有一些做好的 Mixin，可以让开发者直接拿过来使用，例如

TemplateResponseMixin 和 SingleObjectMixin 等，甚至还有 django-braces（http://django-braces.readthedocs.io/en/latest/index.html）提供更多唾手可得的 Mixin，后面我们将使用到。

现在仅仅是开始，好戏在后头。

7.1.4 知识点

1．JSON

JSON 的全称是 JavaScript Object Notation，意思是 JavaScript 对象表示法，是一种独立于语言的轻量级数据交换格式，它使得人们很容易地进行阅读和编写，同时也方便了机器进行解析和生成。

JSON 有以下两种表示结构。

- 结构 1：类 Python 字典模式，例如 obj_json = { "name": "django", "age": 23, "driver": false } 或者 obj_json = { name: "django", age: 23, driver: false }（注意观察区别）。在这种“键值”对的结构中，“键”只能是字符串（可以用引号包裹，也可以不写引号），“值”可以是字符串（string）、数值（number）、布尔值（true、false）、 null、对象（object）和数组（array）。
- 结构 2：类 Python 列表模式，在 JSON 中有一个专有名称——数组，例如 arr_json = ["python", "php", "java"]，数组中的值也必须是字符串、数值、对象、数组、布尔值和 null 类型。

在 JavaScript 的代码中使用 JSON 时，常说“JSON 字符串”，这个字符串并非简单用引号包裹的字符串（如 var lady = "xlaoshi"），而是符合 JSON 格式的 JavaScript 字符串，例如 var lady = "{ age:23, name:'xlaoshi', site:'itdiffer.com' }"。

如何用 Python 来操作 JSON 字符串在《跟老齐学 Python：轻松入门》中有详细阐述，请读者参阅。下面要演示如何在 Django 的视图函数中返回 JSON 值，代码如下。

```
import json
from django.http import HttpResponse

def foo_view(request):
    data = {'name': 'xlaoshi', 'web': 'django'}
    return HttpResponse(json.dumps(data), content_type='application/json')
```

如果使用本章开始的基于类的视图，比如响应前端的 POST 请求，要返回 JSON 值，可以使用 django-braces 提供的第三方 Mixin 解决此问题（关于 django-braces，可参考下一节内容），代码如下。

```
from braces.views import CsrfExemptMixin, JsonRequestResponseMixin
class FooView(CsrfExemptMixin, JsonRequestResponseMixin):
    def post(self, request, *args, **kwargs):
        data = {'name': 'xlaoshi', 'web': 'django'}
        return self.render_json_response(data)
```

2．文档导读

（1）*django-jsonview*，https://github.com/jsocol/django-jsonview。

（2）*Working with JSON and Django*，https://godjango.com/blog/working-with-json-and-django。

（3）*Request and response objects*，https://docs.djangoproject.com/en/2.1/ref/request-response。

7.2　管理课程标题

前面对“基于类的视图”有了初步了解，接下来实现课程标题的管理，如图 7-4 所示。

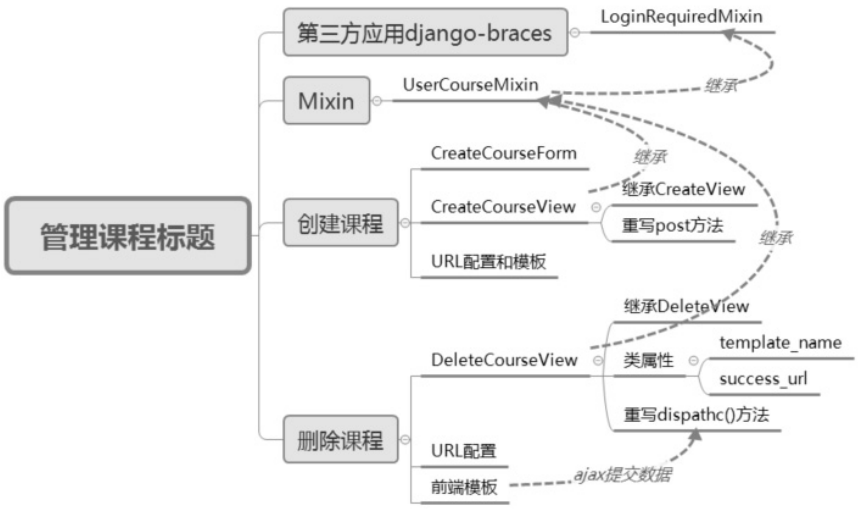

图 7-4　知识技能导图

7.2.1　判断用户是否登录

当用户登录后，进入“后台管理”界面，现在已经有了“文章管理”和“图片管理”功能，要增加“课程管理”功能。编辑./templates/article/header.html 文件，找到导航中的适当位置，增加如下代码。

```
<li><a href="{% url 'course:manage_course' %}">课程管理</a></li>
```

相应地，需要在./course/urls.py 文件中增加如下 URL 配置（不要忘记，在本文件中要引入 ManageCourseListView 类）。

```
path('manage-course/', ManageCourseListView.as_view(), name="manage_course"),
```

在 ManageCourseListView 类中已经声明了 template_name = 'course/manage/manage_course_list.html'，于是创建此文件，输入如下代码。

```
{% extends "article/base.html" %}

{% block title %}manage courses{% endblock %}

{% block content %}
```

```
<div>
    <div class='text-right'><button type="button" class="btn btn-primary">添加课程
</button></div>
    <table class="table table-hover" style="margin-top:10px">
     <tr>
          <td>序号</td>
            <td>课程标题</td>
            <td>发布日期</td>
          <td>操作</td>
     </tr>
     {% for course in courses %}
     <tr id={{ forloop.counter }}>
          <td>{{ forloop.counter }}</td>
          <td>{{ course.title }}</td>
          <td>{{ course.created|date:"Y-m-d" }}</td>
          <td>
              <a name="edit" href="#"><span class="glyphicon glyphicon-pencil"></span>
</a>
              <a nane="delete" href="#"><span class="glyphicon glyphicon-trash"
style="margin-left:20px;"></span></a>
          </td>
     </tr>
     {% endfor %}
    </table>
</div>
{% endblock %}
```

与上述代码显示效果有关的文件./templates/article/leftslider.html 也顺便修改一下，增加与本页面内容相符的左边功能，写法仿照以前的，代码如下。

```
<hr>
<div class="text-center" style="margin-top: 5px;">
    <p><h4>课程管理</h4></p>
    <p><a href="{% url 'course:manage_course' %}">课程管理</a></p>
</div>
```

保存文件之后，在确保 Django 服务运行的情况下，通过打开登录用户名称的下拉菜单，进入“后台管理”界面，可以查看“课程管理”导航，效果如图 7-5 所示。

图 7-5　后台的课程管理

貌似登录用户能够查看到自己的课程标题列表了，但是，读者在编写视图文件中的类时，是不是感觉有点奇怪？当初用视图函数实现类似功能时，会在视图函数前面使用一个装饰器函数，用来判断用户是否处于登录状态，如果没有登录，就转到登录页面，而上面所写的基于类的视图中，没有实现类似功能的代码。

如果用户未登录，访问 http://localhost:8000/course/manage-course/，会得到如图 7-6 所示的错误页面，没有转到登录页面。

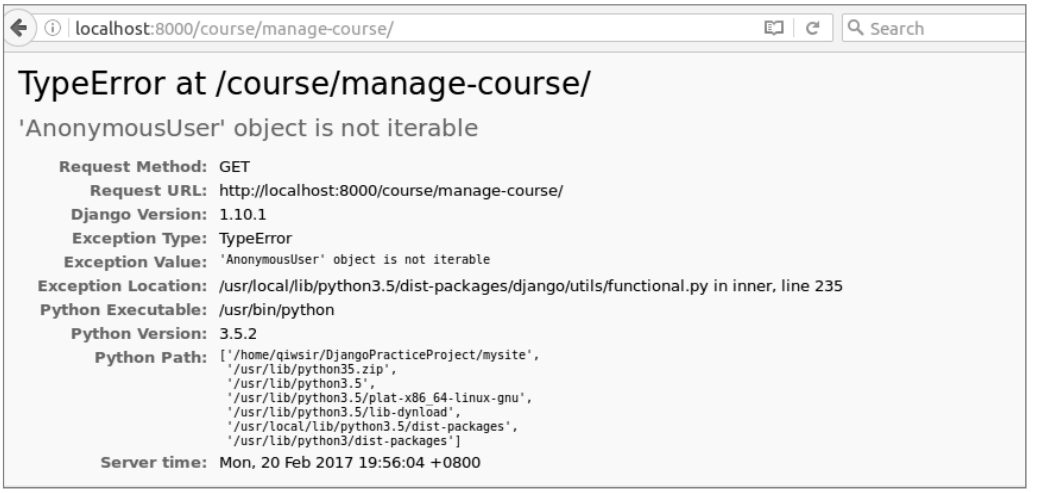

图 7-6　未登录用户访问后的报错页面

要解决上述问题，还要继续对视图文件中的类进行修改。不仅仅是本功能，还有很多其他功能，都要求用户处于登录状态。那么，按照前面的思想，最好写一个 Mixin，在每个有关的类中继承。

对于这种常用的东西，往往有人帮我们解决，程序员也可以继承别人的智慧和成果，“站在巨人的肩膀上”，看得更远，开发更快。

Django-braces 是第三方的应用，里面包含了一些 Django 开发中常用的 Mixin，其官方网站是 https://django-braces.readthedocs.io/en/latest/。

首先安装 Django-braces，代码如下。

```
$ pip3 install django-braces
```

然后编辑./course/views.py 视图文件，在文件头部引入 LoginRequiredMixin，代码如下。

```
from braces.views import LoginRequiredMixin
```

接下来对 UserCourseMixin 类进行重写，代码如下。

```
class UserCourseMixin(UserMixin, LoginRequiredMixin):    #①
    model = Course
    login_url = "/account/login/"    #②
```

与原来不同的地方是语句①和语句②，语句①中增加了一个继承的类 LoginRequireMixin，语句②是新增加的，声明了用户登录的 URL。

其他代码不变。上述修改之后，重新启动 Django 服务，在用户未登录状态下访问 http://localhost:8000/course/manage-course/，就会转到登录页面了，如图 7-7 所示（注意观察图中的 URL）。

图 7-7　转到了登录页面

7.2.2　创建课程

前面以初步了解“基于类的视图”为目的，顺便完成了显示课程标题列表的功能，现在就要完成“创建课程”的功能了，会涉及表单类。根据以往经验，创建./course/forms.py 文件，其表单类代码如下。

```
from django import forms
from .models import Course

class CreateCourseForm(forms.ModelForm):
    class Meta:
        model = Course
        fields = ("title", "overview")
```

这些代码都是熟悉的，没有新知识。紧接着就在./course/views.py 文件中编写处理 GET 和 POST 请求的类，代码如下。

```
from django.views.generic.edit import CreateView
from django.shortcuts import redirect
from .forms import CreateCourseForm

class CreateCourseView(UserCourseMixin, CreateView):    #①
    fields = ['title', 'overview']    #②
    template_name = 'course/manage/create_course.html'

    def post(self, request, *args, **kargs):    #③
        form = CreateCourseForm(data=request.POST)
        if form.is_valid():
            new_course = form.save(commit=False)
            new_course.user = self.request.user
```

```
            new_course.save()
            return redirect("course:manage_course")    #④
        return self.render_to_response({"form":form})    #⑤
```

在文件中完成各种引入，语句①的继承对象列表中包括 CreateView（一个通用视图类），当用户以 GET 方式请求时，即在页面中显示表单，CreateView 就是完成这个作用的类，只要继承它，就不需要写 get()方法了。语句②是声明在表单中显示的字段。

语句③专门处理以 POST 方式提交的表单内容，处理方法与以往的方法一样。要注意语句④，当表单内容被保存后，将页面转向指定位置。

语句⑤是在表单数据检测不通过时，让用户重新填写，注意这里没有使用 render()，而是使用了实例的 render_to_response()方法（self.render_to_response({"form":form})），它与 render()在形式上稍有差别，读者是否可以通过某种方式找出其差别？比如 Google 一下。

完成上面的代码后，就可以在./course/urls.py 文件中配置 URL 了（先引入 CreateCourseView 类），代码如下。

```
path('create-course/', CreateCourseView.as_view(), name="create_course"),
```

最后就剩下创建前端模板了。创建./templates/course/manage/create_course.html 文件，代码如下。

```
{% extends "article/base.html" %}
{% block title %}create course{% endblock %}
{% block content %}
<div style="margin-left:100px;margin-top:10px;">
  <form action="." method="post">{% csrf_token %}
    {{ form.as_p }}
    <input type="submit" value="Create Course">
  </form>
</div>
{% endblock %}
```

这里没有使用 JavaScript，读者可以以最朴素的方式快速地练习本节内容，并且代码中没有使用新的知识。

为了让“课程列表”中的“添加课程”按钮有效，还需要编辑./templates/course/manage/manage_course_list.html 文件中按钮部分的代码，代码如下。

```
<div class='text-right'><a href="{% url 'course:create_course' %}"><button type="button"
class="btn btn-primary">添加课程</button></a></div>
```

上述代码较原来增加了<a>超链接，链接的目标是 create_course。

重启 Django 服务，访问 http://localhost:8000/course/manage-course/，如果没有登录，可以先登录，然后单击页面中的“添加课程”按钮，效果如图 7-8 所示。

图 7-8　添加课程标题

填写表单，然后单击 Create Course 按钮，跳转到课程列表页。

读者不妨测试一下，如果有一个表单没有填写，那么系统就会给出提示。因为在创建 Course 数据模型类时，没有说明哪一项可以为空，所以都是必填项，如图 7-9 所示。

图 7-9　填写表单的提示信息

“创建课程”的功能就这样实现了。读者应该重点理解 CreateView 类，有了它就不用写 get() 的有关方法了，并且在类中可以重写 post()方法，用于处理得到的表单数据。

对于“课程管理”而言，除创建外，还有删除和编辑操作。

7.2.3　删除课程

细心的读者在“课程列表”中看到了两个操作图标，其中由“垃圾桶”图标表示的就是删除操作。在本书前面的内容中，曾经通过编写前端的 JavaScript 脚本，并以 POST 方式向服务器发送请求，从而实现删除功能。在这里，读者当然可以用以前学过的方式实现删除，但是因为现在学习的是基于类的视图，所以应该用不同于以往的方式实现。

在./course/views.py 文件中，引入 DeleteView，类似于 CreateView，也是 Django 内置的基于类的视图，代码如下。

```
from django.views.generic.edit import CreateView, DeleteView
from django.urls import reverse_lazy
```

继续在本文件中编写如下代码。

```
class DeleteCourseView(UserCourseMixin, DeleteView):    #①
    template_name = 'course/manage/delete_course_confirm.html'
    success_url = reverse_lazy("course:manage_course")
```

在 django.views.generic.edit 中，不仅有 CreateView 类，还有 DeleteView 类、UpdateView 类等。语句①中的类继承 DeleteView 类后，后续代码就不需要重复删除动作了，只需要声明确认删除的模板 template_name 和删除完成之后的界面 success_url 即可。

在./course/urls.py 文件中，用下面的方式配置 URL。

```
from .views import DeleteCourseView

urlpatterns = [
    #其他 URL 配置略
    path('delete-course/<int:pk>/', DeleteCourseView.as_view(),name="delete_course"),
]
```

在默认状态下，DeleteView 类接收以 pk 或者 slug 作为参数传入的值，并且通过 GET 方式访问一个删除的确认页面，然后以 POST 方式提交删除表单，才能完成删除（读者或许对这个流程有异议，暂且保留）。按照上述流程，以 GET 方式访问的地址就是上述配置的 URL，其模板页面即为 template_name 所规定的页面，下面的代码就是./templates/course/manage/delete_course_confirm.html。

```
{% extends "article/base.html" %}
{% block title %}delete course{% endblock %}
{% block content %}
<div>
    <form action="" method="post">{% csrf_token %}
    <p>Are you sure you want to delete "{{ object.title }}"?</p>    #②
    <input type="submit" value="Confirm" />
</form>
</div>
{% endblock %}
```

因为在 DeleteCourseView 类中没有使用 context_object_name 声明渲染模板的变量名称，所以在模板中像语句②这样使用默认的变量 object 是不提倡的。这里如此使用，一是为了告知读者，二是因为此模板简单，如此处理也未尝不可。

最后要做的是在./templates/course/manage/manage_course_list.html 文件中为“垃圾桶”图标做好超链接，实现单击它进入删除确认页面中，代码如下。

```
<a nane="delete" href="{% url 'course:delete_course' course.id %}" ><span class="glyphicon glyphicon-trash" style="margin-left:20px;"></span></a>
```

确保 Django 服务已经运行，在浏览器中访问 http://localhost:8000/course/manage-course/，查看该用户已有的课程名称列表，单击“删除”图标，进入删除确认页面。

单击 Confirm 按钮，如图 7-10 所示，即可删除当前课程标题，并转到课程标题列表页面。

图 7-10　确定删除操作页面

至此，完成了删除功能。

但是，细思量，不满意。

如果上述删除操作用前端的 JavaScript 完成，其流程是：①单击“删除”图标；②弹出确认提示框；③单击“确认”按钮；④提交数据；⑤返回结果并跳转/刷新页面。这样的流程才是理想的删除流程，而非上面那样——写一个 Confirm 功能页面，是比较蹩脚的做法。

为了实现上述的删除流程，必须重写./course/views.py 中的 DeleteCourseView 类，重写之后的代码如下。

```
#以下两行是配合 DeleteCourseView 类需要在文件顶部引入的模块
from django.http import HttpResponse
import json

class DeleteCourseView(UserCourseMixin, DeleteView):
    #template_name = 'course/manage/delete_course_confirm.html'
    success_url = reverse_lazy("course:manage_course")

    def dispatch(self, *args, **kwargs):    #③
        resp = super(DeleteCourseView, self).dispatch(*args, **kwargs)    #④
        if self.request.is_ajax():    #⑤
            response_data = {"result": "ok"}
            return                                HttpResponse(json.dumps(response_data),
content_type="application/json")
        else:
            return resp
```

语句③重写了 DeleteView 类中的 dispatch()方法，在这个方法中首先执行语句④。原本在 DeleteView 类中执行 dispatch()方法后，会实现 URL 的转向，但是在此指令发送给前端之前，通过语句⑤进行判断，如果是 Ajax 方法提交过来的数据，就直接反馈 HttpResponse 对象给前端，前端的 JavaScript 函数得到反馈结果，这样就完成了删除和页面的刷新。为此，在前端模板中就要嵌入 JavaScript 代码。下面是重写了./templates/course/manage/manage_course_list.html 之后的代码。

```
{% extends "article/base.html" %}
{% load staticfiles %}
{% block title %}manage courses{% endblock %}
{% block content %}
<div>
    <div class='text-right'>
```

```
        <a href="{% url 'course:create_course' %}">
            <button type="button" class="btn btn-primary">添加课程</button>
        </a>
    </div>
    <table class="table table-hover" style="margin-top:10px">
     <tr>
          <td>序号</td>
            <td>课程标题</td>
            <td>发布日期</td>
          <td>操作</td>
     </tr>
     {% for course in courses %}
     <tr id={{ forloop.counter }}>
          <td>{{ forloop.counter }}</td>
          <td>{{ course.title }}</a></td>
          <td>{{ course.created|date:"Y-m-d" }}</td>
          <td>
              <a name="edit" href="javascript:">
                  <span class="glyphicon glyphicon-pencil"></span>
              </a>
              <a class="delete" name="delete"
                     href="{% url 'course:delete_course' course.id %}" >
                  <span class="glyphicon glyphicon-trash" style="margin-left:20px;">
                  </span>
              </a>
          </td>
     </tr>
     {% endfor %}
    </table>
</div>
<script type="text/javascript" src='{% static "js/jquery.js" %}'></script>
<script type="text/javascript">
function getCookie(name) {
    var cookieValue = null;
    if (document.cookie && document.cookie != '') {
        var cookies = document.cookie.split(';');
        for (var i = 0; i < cookies.length; i++) {
            var cookie = jQuery.trim(cookies[i]);
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) == (name + '=')) {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
}

$(document).ready(function() {
    var csrftoken = getCookie('csrftoken');
    function csrfSafeMethod(method) {
```

```
        return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
    }
    $.ajaxSetup({
        crossDomain: false, // obviates need for sameOrigin test
        beforeSend: function(xhr, settings) {
            if (!csrfSafeMethod(settings.type)) {
                xhr.setRequestHeader("X-CSRFToken", csrftoken);
            }
        }
    });

    var onDelete = function(){
        alert("delete it?");
        $.post(this.href, function(data) {
            if (data.result == "ok"){
              window.location.reload();
            } else {
                alert("sth wrong");
            }
        }).fail(function() {
            alert("error");
        });
        return false;
    }

    $(".delete").click(onDelete);
})
</script>
{% endblock %}
```

在 JavaScript 部分，用一种新的方式解决 CSRF 问题，从而实现用 POST 方式向后端提交数据，请读者认真阅读上述代码，这是另外一种解决问题的方法。当然，上述代码在实现效果上比较朴素，读者完全可以在上述基础上对某些效果进行优化。

完成之后，访问 http://localhost:8000/course/manage-course/进行测试。

如图 7-11 所示，单击 OK 按钮即可删除一个课程并重新载入本页面。

图 7-11　删除操作

顺利完成删除功能。

我们已经使用了 django.views.generic.edit 中的 CreateView 类和 DeleteView 类，还有一个 UpdateView 类，这三个类实现了“增改删”常用功能。因为 UpdateView 类的使用方法与其他类雷同，所以此处不再赘述，请读者自行学习，可以参考官方文档（见“知识点”中的“文档导读”）

7.2.4　知识点

1．模型：继承

“继承”是对象的重要特征，在《跟老齐学 Python：轻松入门》一书中对 Python 类的继承有详细阐述，特别建议读者温习，有助于对 Django 中关于模型继承的理解。

这里所说的 Django 数据模型的继承，不是指前面我们已经看到的数据模型类继承 models.Model，而是在下述情况下所发生的继承。

（1）在多个数据模型类中都有相同的字段（在本项目的各个应用中就有相同的字段），在这种情况下，可以把多个相同的字段抽取出来定义一个类，然后其他的类继承这个类。

（2）在某个应用中已经有一个数据模型，新的数据模型要用到这个数据模型的字段（不需要在新的数据模型中建立与已有数据模型重复的字段），此时也可以使用继承。

（3）如果新的数据模型类相对已有的类只有行为的变化，比如排序方式变化，那么也适合用继承的方法解决。

针对上述三种情况，Django 中提供了三种类型的继承方式。

（1）抽象模型继承

针对第一种情况，可以建立“抽象模型”，即在数据模型类的内部类中声明 abstract = True，则该数据模型即为抽象模型。例如：

```
from django.db import models
class BaseInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True
        ordering = [ ‘age’ ]

class Programmer(BaseInfo):
    lang = models.CharField(max_length=100)

class Boss(BaseInfo):
    ppt = models.BooleanField()
```

数据模型类 Programmer 和 Boss 通过继承 BaseInfo，实现了将 BaseInfo 中的字段和行为继承过来的目的。当实施数据迁移后，在数据库表中，没有 BaseInfo 所生成的表，只有 Programmer 和 Boss 对应的表，在其数据库表的结构中，包含了 name 和 age 字段。

在 BaseInfo 中，如果 name = models.ForeignKey(User, on_delete=models.CASCADE, related_name="user_base")，这里使用 ForeignKey，目的是当 BaseInfo 被子类继承之后，每个子类和 User 建立“一对多”的关系。然而，按照上面的方式定义 name 会出问题，Programmer 和 Boss 中都是 related_name="user_base"，如果使用 User 反向查询，则不知所踪了。对此 Django 早有设计，专门解决此问题。

```
class BaseInfo(models.Model):
    name = models.ForeignKey(User, on_delete=models.CASCADE,
                             related_name="%(app_label)s_%(class)s_related")
    #省略其他代码
```

“%(app_label)s”和“%(class)s”都是占位符，从名称上就能猜到，前者对应的是应用名称，后者对应的是子类的名称。这样子类中的 related_name 就不冲突了，而且跨应用继承也毫无问题。

（2）多表继承

多表继承是针对第二种情况的。

官方文档中有一个非常合适的例子，引用过来，相信读者就能理解了。

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

这种继承，本质上是建立了 Place 和 Restaurant 之间的“一对一”关系，就如同在子类中设置了一个 OneToOneField 类型的字段。

当完成数据迁移后，两个数据模型都会有对应的数据库表。

（3）代理模型

代理模型就是子类通过自定义的方法或者重写内部类 Meta 的方式，仅改变模型的行为。例如：

```
class Student(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

class Teacher(Student):
    class Meta:
        proxy = True
        ordering = ['name']

    def teacher_age(self):
        return self.age + 17
```

数据模型类 Teacher 继承了 Student，子类没有声明任何字段。在 Meta 内部类中，通过 proxy = True 说明 Teacher 相对 Student 的关系，并且 Teacher 的排序规则为 ordering = ['name']，这就是行为的变化。类的行为还体现在方法上，后面定义的 teacher_age()方法就是如此。

2．文档导读

（1）*Model inheritance*，https://docs.djangoproject.com/en/2.1/topics/db/models/#model-inheritance。

（2）*Class-based views*，https://docs.djangoproject.com/en/2.1/topics/class-based-views。

（3）*Using mixins with class-based views*，https://docs.djangoproject.com/en/2.1/topics/class-based-views/mixins。

7.3　发布和学习课程内容

对于课程内容，使用诸如 Markdown 编辑器这样的在线编辑器就可以实现，但是这么做就与前面发布文章时用的方法类似了。

为了学习更多的技能，这里换一种发布方式，假设所发布的课程内容是视频，并且可以携带附件，于是发布课程内容的界面应该包括以下几项。

- 内容标题。
- 所属课程。通过下拉菜单选择本内容属于哪一个课程标题。
- 上传视频。
- 视频描述。
- 本课附件。

根据上述要求，开始制作发布和学习课程内容的功能，如图 7-12 所示。

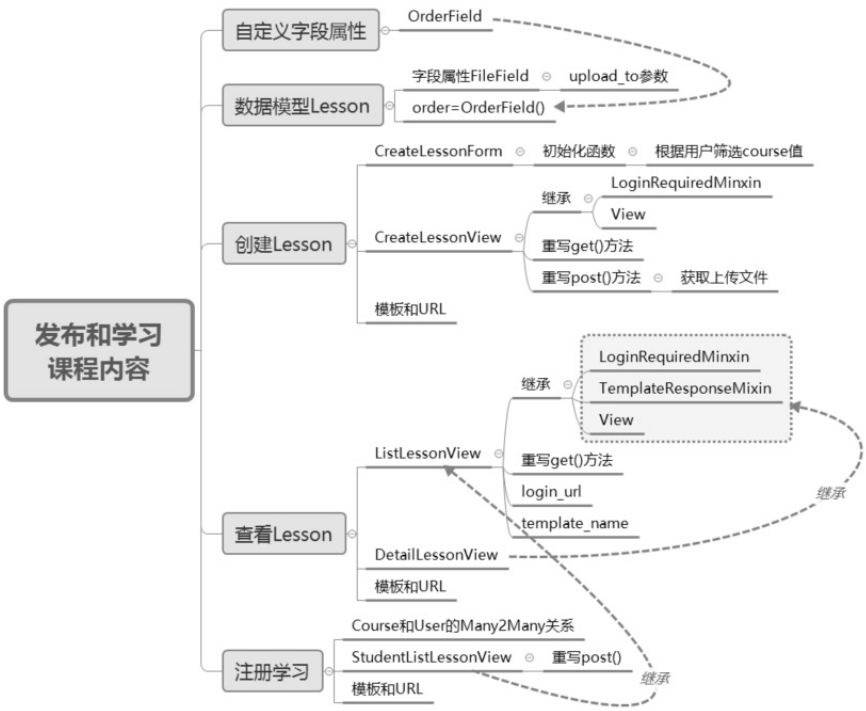

图 7-12　知识技能导图

7.3.1 课程内容的数据模型

根据课程内容的要求和已经拥有的经验，可以编写出课程内容的数据模型（编辑./course/models.py 文件），代码如下。

```
class Lesson(models.Model):
    user = models.ForeignKey(User,  on_delete=models.CASCADE,
                             related_name='lesson_user')
    course = models.ForeignKey(Course,  on_delete=models.CASCADE,
                               related_name='lesson')
    title = models.CharField(max_length=200)
    video = models.FileField(upload_to="videos")    #①
    description = models.TextField(blank=True)
    attach = models.FileField(blank=True, upload_to="attachs")    #②
    created = models.DateTimeField(auto_now_add=True)
```

在 Lesson 类中，语句①和语句②是用来接收上传的视频和附件的，其中用参数 upload_to 规定所上传文件的存储路径，这与在第 5 章中学习过的收集图片是一样的。但是深入思考，会发现友好性稍差，最好是把用户上传的文件存放到用户自己的目录中，于是可以在这个类前面增加一个函数，代码如下。

```
def user_directory_path(instance, filename):
    return "courses/user_{0}/{1}".format(instance.user.id, filename)
```

这个函数接收两个参数，instance 引用的是 Lesson 类实例，filename 则为得到的文件名。Lesson 类中语句①和语句②的 upload_to 就可以通过这个函数确定新的文件存储路径，将语句①和语句②分别修改为如下形式。

```
video = models.FileField(upload_to=user_directory_path)
attach = models.FileField(blank=True, upload_to=user_directory_path)
```

每个课程标题下会有一个或者多个课程内容，这里就涉及到对课程内容排序的问题了。虽然可以用自增的 id 作为排序字段，但是不能显示该内容在相应课程标题下的顺序（序号就不连贯了，当然在显示的时候，也能够处理成连贯的）。下面讲解一种方法，实现在数据库中保存该内容在相应课程标题下的序号。

这就是学习“自定义字段属性”的理由，根据此理由，创建一个自定义的字段属性。

在./course 目录中创建 fields.py 文件，并在其中添加如下代码。

```
from django.db import models
from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):    #③
    def __init__(self, for_fields=None, *args, **kwargs):
        self.for_fields = for_fields
        super(OrderField, self).__init__(*args, **kwargs)

    def pre_save(self, model_instance, add):    #④
        if getattr(model_instance, self.attname) is None:    #⑤
            try:
                qs = self.model.objects.all()    #⑥
```

```
                if self.for_fields:
                    query = {field: getattr(model_instance, field)
                                                for field in self.for_fields}   #⑦
                    qs = qs.filter(**query)    #⑧
                last_item = qs.latest(self.attname)    #⑨
                value = last_item.order + 1    #⑩
            except ObjectDoesNotExist:
                value = 0
            setattr(model_instance, self.attname, value)    #⑪
            return value
        else:
            return super(OrderField, self).pre_save(model_instance, add)
```

自定义的字段属性在本质上就是一个类。语句③定义了类的名称，并且继承 models.PositiveIntegerField。这里所定义的 OrderField 是要得到对象排序的序号，其值为整数，所以继承 models.PositiveIntegerField 是合适的。

在 Django 的字段属性中，都继承了 Field 类，pre_save()就是 Field 类中的一个方法，它的源码是（https://docs.djangoproject.com/en/2.1/_modules/django/db/models/fields/#Field.pre_save）：

```
def pre_save(self, model_instance, add):
    """
    Returns field's value just before saving.
    """
    return getattr(model_instance, self.attname)
```

pre_save()方法的作用是在保存之前对数值进行预处理，在具体的某个字段属性中，因为特殊需要，常常将 Field 类中的这个方法重写。例如前面已经用到的 DateTimeField，曾经在数据模型类中有这样的属性（字段）created = models.DateTimeField(auto_now_add=True)或者 updated = models.DateTimeField(auto_now=True)，在 DateTimeField 中就重写了 pre_save()方法，对时间进行预处理（完整代码请参阅 https://docs.djangoproject.com/en/2.1/_modules/django/db/models/fields/#DateTimeField），使得我们不需要单独显式地把保存的当前时间写出来，而是在调用实例的 save()方法之后，自动完成时间的写入保存。

语句④也是通过重写 pre_save()方法，最终将实例的序号记录下来。从前面的 Field.pre_save()方法可以得知，参数 model_instance 和 add 是与祖先类保持一致的，这样的写法友好性更强。model_instance 引用的是实例，add 为该实例是否第一次被保存。

语句⑤中的 getattr()是 Python 的内建函数，它能够返回一个对象属性的值（关于此函数的详细说明请参阅 https://docs.python.org/3/library/functions.html#getattr），getattr(model_instance, self.attname)中的 self.attname 也是在 Fields 类里面规定的一个参数。在 Fields 类中有如下两个方法（https://docs.djangoproject.com/en/2.1/_modules/django/db/models/fields/#Field）。

```
def get_attname_column(self):
    attname = self.get_attname()
    column = self.db_column or attname
    return attname, column

def set_attributes_from_name(self, name):
    if not self.name:
```

```
            self.name = name
            self.attname, self.column = self.get_attname_column()
            self.concrete = self.column is not None
        if self.verbose_name is None and self.name:
            self.verbose_name = self.name.replace('_', ' ')
```

所以在语句⑤中也使用 self.attname 参数，判断当前对象（实例）中是否有某个属性（字段），如果有，就执行 else 分支，调用父类的 pre_save()方法，但不会在数据库中增加记录；否则就执行 try … except …语句，在 try 中主要计算新增一条数据后的序号。

语句⑥得到当前实例的所有记录。

语句⑦得到字段列表中的属性名称（字段名称）在本实例中是否存在字典。

语句⑧根据语句⑤中的数据对语句④的结果进行筛选。

语句⑨根据 self.attname 得到经过语句⑥筛选之后的记录中的最后一条。

语句⑩对当前实例进行序号的编排。

经过上面的工作，最后用语句⑪在相应的字段上记录本实例的序号，返回该序号值并通过 pre_save()自动保存。

如果读者阅读上面的内容后，感觉没有理解，可以暂且搁置，继续向下进行，然后回头反复阅读，就能体会出上面代码的含义了。

完成了自定义属性的设置，下面就要在数据模型类中应用它。

在./course/models.py 中引入刚才完成的 OrderField 类，代码如下。

```
from .fields import OrderField
```

在 Lesson 类中增加如下内容。

```
class Lesson(models.Model):
    #其他字段
    order = OrderField(blank=True, for_fields=['course'])

    class Meta:
        ordering = ['order']

    def __str__(self):
        return '{}.{}'.format(self.order, self.title)
```

order 用来存储某内容在相应的课程标题 Course 中的序号（序号从 0 开始）。最后，请对照 Lesson 类，其全貌应该是下面这样的。

```
class Lesson(models.Model):
    user = models.ForeignKey(User, related_name='lesson_user')
    course = models.ForeignKey(Course, related_name='lesson')
    title = models.CharField(max_length=200)
    video = models.FileField(upload_to=user_directory_path)
    description = models.TextField(blank=True)
    attach = models.FileField(blank=True, upload_to=user_directory_path)
    order = OrderField(blank=True, for_fields=['course'])
```

```
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['order']

    def __str__(self):
        return '{}.{}'.format(self.order, self.title)
```

数据库模型建立好了，就可以进行数据的迁移操作，依次执行 python3 manage.py makemigrations course 和 python3 manage.py migrate 指令即可。

完成数据模型的构建和数据迁移指令后，请读者自行查看数据库表的结构，从而得到更直观的体验。

7.3.2　课程内容的表单类

根据前述需求，提交课程内容时一定会用到表单，所以构建表单类是必不可少的，其代码如下（编辑./course/forms.py 文件）。

```
from .models import Course, Lesson

class CreateLessonForm(forms.ModelForm):
    class Meta:
        model = Lesson
        fields = ['course', 'title', 'video', 'description', 'attach']
```

这些都是老生常谈的东西了，不足为奇，但面对貌似熟悉的东西时，更要小心行事。如果仅按照上面的方法来写表单，就会遇到一个“小坑”。本项目中是多个用户，每个用户都可以创建自己的课程标题（course），如果按照上面表单类中的 fields 声明，必然会列出所有用户创建的 course，而这不是功能所需要的。功能的要求是每个用户只能看到自己所设置的课程标题。对于这个需求，可以在视图中解决，而这里要讲解一种在表单类中解决的方法，就是重写__init__()初始化函数，代码如下。

```
from .models import Course, Lesson

class CreateLessonForm(forms.ModelForm):
    class Meta:
        model = Lesson
        fields = ['course', 'title', 'video', 'description', 'attach']
    def __init__(self, user, *args, **kwargs):    #①
        super(CreateLessonForm, self).__init__(*args, **kwargs)
        self.fields['course'].queryset = Course.objects.filter(user=user)    #②
```

请注意语句①中的参数多了一个 user，通过这个参数传入当前用户，并用于语句②，筛选出当前用户的 course 值。

7.3.3　课程内容的视图

与课程内容有关的数据模型和表单都做好了，接下来就要进行视图部分的代码编写，依然是写基于类的视图。按照前面的经验，这个类依然可以继承 CreateView，但为了学习技能，改

用其他方法，先看下面的代码（编辑./course/views.py 文件）。

```
from django.shortcuts import redirect, render
from django.views import View    #①
from .forms import CreateLessonForm
from .models import Lesson

class CreateLessonView(LoginRequiredMixin, View):    #②
    model = Lesson
    login_url = "/account/login/"

    def get(self, request , *args, **kwargs):    #③
        form = CreateLessonForm(user=self.request.user)    #④
        return render(request, "course/manage/create_lesson.html", {"form":form})

    def post(self, request, *args, **kwargs):    #⑤
        form = CreateLessonForm(self.request.user, request.POST, request.FILES)    #⑥
        if form.is_valid():
            new_lesson = form.save(commit=False)
            new_lesson.user = self.request.user
            new_lesson.save()
            return redirect("course:manage_course")
```

语句①所引入的类在有的程序中也写成 from django.views.generic.base import View，所引入的 View 类是所有基于类的视图的基类，前面已经学习过的 CreateView 等都继承了它。这里语句②不再继承 CreateView，但必须继承 View。

在 View 类中没有默认的 get()和 post()方法，因此需要动手写这两个方法，以响应前端发出的两种方式的请求。

注意观察语句③中的参数，因为 get()是当前所在类中的一个方法，所以第一个参数必须是 self。另外，get()还要接收网站前端所提交的数据，即 HttpRequest 对象，所以第二个参数使用 request，这与写视图函数一样。后面的参数是什么意思呢？按照对 Python 函数中的参数的理解，读者理应明白，还可以参考《跟老齐学 Python：轻松入门》中的相关内容。

语句④创建了表单类的实例，注意在表单类 CreateLessonForm 中，已经重写了初始化函数__init__()，并且增加了一个参数 user，所以在实例化时需要传入 user 值。

语句⑤响应用户提交的表单，其参数含义同前。读者要注意观察此方法中的语句⑥，因为提交的表单中有上传的文件，所以必须传入 request.FILES。

根据 get()方法中约定的模板，创建./templates/course/manage/create_lesson.html 文件，并输入如下代码。

```
{% extends "article/base.html" %}
{% block title %}create lesson{% endblock %}
{% block content %}
<div style="margin-left:200px;margin-top:10px;">
  <form enctype="multipart/form-data" action="." method="post" class="form-horizontal">
{% csrf_token %}
    <div style="margin-top: 10px;">
        <div class="col-md-2 text-left"><span>内容标题:</span></div>
```

```
        <div class="col-md-10 text-left">{{form.title}}</div>
    </div>
    <div style="margin-top: 20px;">
        <div class="col-md-2 text-left"><span>所属课程:</span></div>
        <div class="col-md-10 text-left">{{form.course}}</div>
    </div>
    <div style="margin-top: 20px;">
        <div class="col-md-2 text-left"><span>上传视频:</span></div>
        <div class="col-md-10 text-left">{{form.video}}</div>
    </div>
    <div style="margin-top: 20px;">
        <div class="col-md-2 text-left"><span>视频描述:</span></div>
        <div class="col-md-10 text-left">{{form.description}}</div>
    </div>
    <div style="margin-top: 20px;">
      <div class="col-md-2 text-left"><span>本课附件:</span></div>
      <div class="col-md-10 text-left">{{form.attach}}</div>
    </div>
    <div style="margin-top: 20px;" class="text-center">
      <input type="submit" value="Create Lesson">
    </div>
  </form>
</div>
{% endblock %}
```

此模板文件的代码无须过多解释，相信读者已经熟悉。最后要在./course/urls.py 文件中配置 URL，增加如下内容。

```
from .views import CreateLessonView   #文件顶部引入
#以下是 URLconf 的值
path('create-lesson/', CreateLessonView.as_view(), name="create_lesson"),
```

为了显示完整，在./templates/article/leftslider.html 中增加如下链接代码。

```
<p><a href="{% url 'course:create_lesson' %}">内容发布</a></p>
```

重新启动 Django 服务，在浏览器中访问 http://localhost:8000/course/create-lesson/，如果没有登录，会跳转到登录页面进行登录；对于已经登录的用户，显示效果如图 7-13 所示。

图 7-13　发布课程内容

在这个页面中可以发布课程内容。如果读者发布了课程内容，数据库会把相关信息保存下来，此时数据库内容如下。

```
sqlite> select * from course_lesson;
id          title          video                description   attach      created                    order       course_id   user_id
----------  -------------  -------------------  ------------  ----------  -------------------------  ----------  ----------  --------
1           what is python  courses/python01.mp4  leran python              2018-06-18 08:24:17.532193  0           6           8
```

从数据库中保存的信息可以看出，我们曾经自定义的字段属性 OrderField 已经发挥了作用，并且也能看到上传的视频和附件的保存信息，不妨再到相应的目录中看看是否存在该文件。

至此，已经能够发布每个课程标题之下的课程内容了，但课程内容发布之后，如何查看和管理呢？

7.3.4 查看课程内容

在用户的管理界面中查看课程标题下面的课程内容，基本功能需求如下。

- 在“课程管理”列表的“操作”列中增加一个“查看”图标；
- 单击“查看”图标能够列出该课程标题下的所有课程内容标题；
- 在“课程内容标题”列表页面，单击每个标题名称，就可以查看该课程内容的详细信息。

下面按照功能需求依次开发。

1．课程内容标题列表

依然可以使用 ListView 完成课程内容标题列表的显示功能，这个操作建议读者自行完成（一定要自己完成，不要略过）。完成之后再继续阅读，下面用一种不同的方式实现此功能。

还是编写基于类的视图，代码如下（下面的代码依然写到./course/views.py 文件中）。

```
from django.shortcuts import get_object_or_404
from django.views.generic.base import TemplateResponseMixin    #①

class ListLessonsView(LoginRequiredMixin, TemplateResponseMixin, View):    #②
    login_url = "/account/login/"
    template_name = 'course/manage/list_lessons.html'    #③

    def get(self, request, course_id):    #④
        course = get_object_or_404(Course, id=course_id)    #⑤
        return self.render_to_response({'course':course})    #⑥
```

语句①引入了 TemplateResponseMixin 类，它提供了一种模板渲染的机制，在子类中，可以指定模板文件和渲染数据。语句②所定义的 ListLessonView 类继承 TemplateResponseMixin，利用语句③定义模板文件。

语句④是相应前端 GET 请求的方法，因为要识别课程标题，所以传入了参数 course_id。语句⑤根据 course_id 得到当前的“课程标题”对象，通过语句⑥将该数据渲染到模板中。render_to_response()就是 TemplateResponseMixin 类的方法。

下面根据语句③所定义的模板创建模板文件（./templates/course/manage/list_lessons.html），并编写相应代码。

```
{% extends "article/base.html" %}
{% block title %}List Lessons{% endblock %}
{% block content %}
<div>
    <div class='text-center'>
      <h1>{{ course.title }}</h1>
        <p>课程内容列表</p>
    </div>
    <table class="table table-hover" style="margin-top:10px">
     <tr>
         <td>序号</td>
           <td>内容标题</td>
           <td>发布日期</td>
     </tr>
     {% for lesson in course.lesson.all %}    #⑦
     <tr id={{ forloop.counter }}>
         <td>{{ forloop.counter }}</td>
         <td>{{ lesson.title }}</a></td>
         <td>{{ lesson.created|date:"Y-m-d" }}</td>
     </tr>
     {% endfor %}
    </table>
</div>
{% endblock %}
```

语句⑦是需要说明的。通过语句⑥传入到模板中的 course 是实例对象，再观察 Lesson 数据模型类，其中有 course = models.ForeignKey(Course, related_name='lesson')，因此可以通过反向查询得到 Lesson 的实例，即 course.lesson.all 为所有 Lesson 实例集合。

模板创建完成了，还要配置 URL，编辑./course/urls.py 文件（先在本文件中引入 ListLessonsView 类）。

```
path('list-lessons/<int:course_id>/', ListLessonsView.as_view(), name="list_lessons"),
```

课程内容的列表如图 7-14 所示。笔者测试的系统中有 Course 的 id 为 6 的记录，因此可以在浏览器的地址栏中输入 http://localhost:8000/course/list-lessons/6/，如果读者测试系统中的 id 值不同，请注意修改，具体 id 值可以在数据库中查看。

图 7-14　课程的内容列表

为了从“课程管理”列表中查看该课程的内容列表，需要再次编辑./templates/course/manage/manage_course_list.html 文件，在“操作”列下面增加如下代码。

```
<a href="{% url 'course:list_lessons' course.id %}"><span class="glyphicon glyphicon-search" style="margin-left:20px;"></span></a>
```

最终显示效果如图 7-15 所示。

图 7-15　查看课程内容列表的入口

单击“查看”图标（放大镜图标）后，就可以浏览相应课程标题所属的课程内容了。

还要提醒读者注意，在上图的“操作”列中，没有带领读者开发“编辑”（笔形图标）功能。“编辑”功能就是单击“编辑”图标即可进入对当前课程标题再编辑的界面。如果读者已经跟随本书将每一步都认真学习了，那么这个功能读者有足够的能力自行开发，请试一试。

2．课程内容详情

显示课程内容详情的方法有很多种，在这里使用的是与前面一样的方法，目的在于重复练习，加深理解。对于部分读者而言，这可能不是最佳的技术方案，如果是这样请略过以下内容。

在./course/views.py 文件中编写 DetailLessonView 类，代码如下。

```
class DetailLessonView(LoginRequiredMixin, TemplateResponseMixin, View):
    login_url = "/account/login/"
    template_name = "course/manage/detail_lesson.html"

    def get(self, request, lesson_id):
        lesson = get_object_or_404(Lesson, id=lesson_id)
        return self.render_to_response({"lesson":lesson})
```

因为是重复练习，所以不再解释代码了，读者可以自己注释一下。下面创建模板文件./templates/course/manage/detail_lesson.html，代码如下。

```
{% extends "article/base.html" %}
{% block title %}Lesson Detail{% endblock %}
{% block content %}
<div>
    <div class='text-center'>
        <h1>{{ lesson.title }}</h1>
        <p>讲师：{{ lesson.user.username }}</p>
```

```
    </div>
    <div>
      {% if lesson.video %}
      <video src="{{ lesson.video.url }}" controls="controls" class="text-center">
</video>
      {% else %}
      <p class="text-center">There is not video lesson.</p>
      {% endif %}
    </div>
    <div>
      {% if lesson.description %}
      <p>{{ lesson.description }}</p>
      {% endif %}
    </div>
    <div>
      {% if lesson.attach %}
      <p><a href="{{ lesson.attach.url }}">{{ lesson.attach.name | cut:"courses/user_" }}
</a></p>
      {% endif %}
    </div>
</div>
{% endblock %}
```

还记得曾经在./mysite/urls.py 文件中有 urlpatterns += static(settings.MEDIA_URL,document_root= settings.MEDIA_ROOT)设置吗？并且曾经在./mysite/settings.py 文件中已经设置好了，代码如下。

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

所以上述模板代码中的{{ lesson.video.url }}才能实现访问静态文件的 URL 功能。

最后配置 URL，在./course/urls.py 文件中增加如下内容（先引入 DetailLessonView）。

```
path('detail-lesson/<int:lesson_id>/', DetailLessonView.as_view(), name="detail_
lesson"),
```

以上设置都完成了，就可以进行调试。比如，在浏览器的地址栏中输入 http://localhost:8000/course/detail-lesson/1/，显示效果如图 7-16 所示。

调试没问题，就可以在课程内容标题列表页做超链接，作为访问详细内容的入口。编辑./templates/course/manage/list_lessons.html 文件，对内容标题部分做超链接，代码如下。

```
{% for lesson in course.lesson.all %}
<tr id={{ forloop.counter }}>
<td>{{ forloop.counter }}</td>
<td><a href="{% url 'course:detail_lesson' lesson.id %}">{{ lesson.title }}</a></td>
<td>{{ lesson.created|date:"Y-m-d" }}</td>
</tr>
{% endfor %}
```

完成之后进行调试。

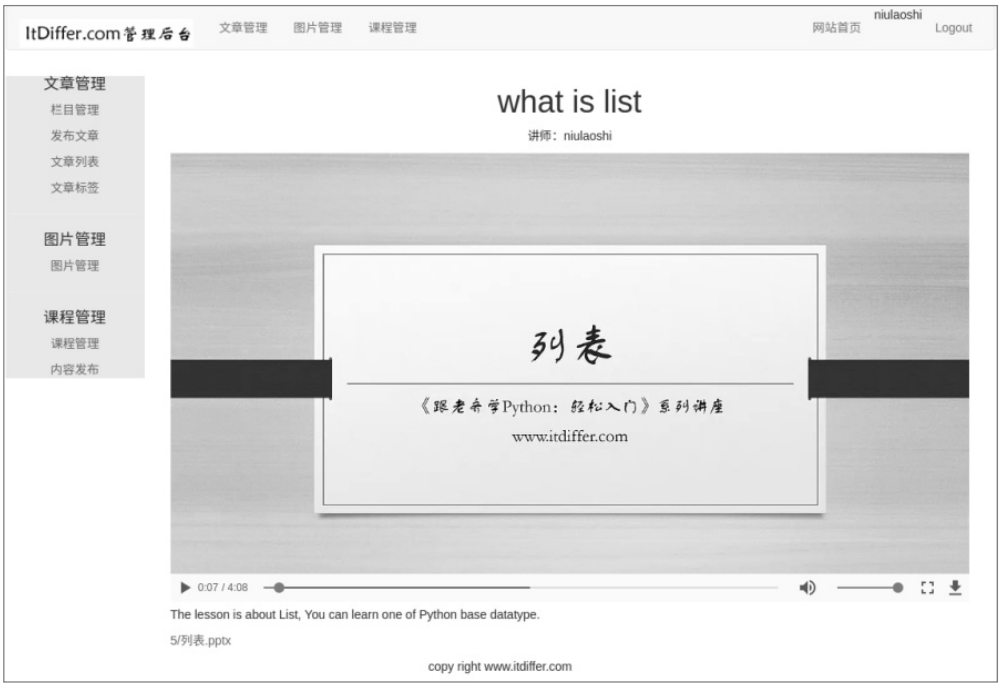

图 7-16　视频内容播放页面

至此，在后台管理端就完成了从课程标题到课程内容的一系列创建和查看功能。现在读者可以审视一下完成的内容，如果还有需要补充的功能，可以利用在本章中已经学习的各种知识再次进行开发和优化——还有很多环节需要进一步完善呢，这是读者施展自己能力的机会。

7.3.5　注册学习课程

用户学习每个课程前都要注册该课程，也就是在数据库中记录用户开始学习该课程了，同时也要记录这个课程的学习者中有该用户。由此可知，课程和用户之间构成了“多对多”的关系。为了在数据模型上体现这种关系，需要编辑./course/models.py 文件中的 Course 数据模型类，增加如下字段。

```
student = models.ManyToManyField(User, related_name="courses_joined", blank=True)
```

数据模型类修改好之后，就可以迁移数据了。完成之后，在数据库中会有所表现，多了一个名为 course_course_student 的表。

```
sqlite> pragma table_info(course_course_student);
cid         name        type        notnull     dflt_value  pk
----------  ----------  ----------  ----------  ----------  ----------
0           id          integer     1                       1
1           course_id   integer     1                       0
2           user_id     integer     1                       0
```

在本章 7.1 节中，访问 http://localhost:8000/course/course-list/就可以得到所有用户发布的课程标题。下面就用前面已经写好的 ListLessonsView 类和 DetailLessonView 类来展示每个课程中的内容标题和详细内容，但是如果真的这么做了，会发现模板文件有点别扭，因为前端浏览的模板文件和管理界面的模板文件有差别，怎么办？

```
class StudentListLessonView(ListLessonsView):    #①
    template_name = "course/slist_lessons.html"    #②
```

在./course/views.py 文件中创建 StudentListLessonView 类，如语句①所示，这个类继承上面已经创建的 ListLessonView 类，因为显示的模板文件有所不同，所以语句②重写模板。是不是越来越体会到基于类的视图带来的便利了？

接下来的工作就是创建模板文件./templates/course/slist_lessons.html，代码如下。

```
{% extends "base.html" %}
{% block title %}Lessons List{% endblock %}
{% block content %}
<div class="container">
  <div class='text-center'>
    <h1>{{ course.title }}</h1>
    <p>课程内容列表</p>
  </div>
  <table class="table table-hover" style="margin-top:10px">
    <tr>
      <td>序号</td>
      <td>内容标题</td>
      <td>概要</td>
      <td>发布日期</td>
    </tr>
    {% for lesson in course.lesson.all %}
    <tr id={{ forloop.counter }}>
      <td>{{ forloop.counter }}</td>
      <td>{{ lesson.title }}</td>
      <td>{{ lesson.description | slice:'70'|linebreaks }}</td>
      <td>{{ lesson.created|date:"Y-m-d" }}</td>
    </tr>
    {% empty %}
    <p>还没有发布具体内容，请等待。</p>
    {% endfor %}
  </table>
</div>
{% endblock %}
```

在./course/urls.py 文件中设置 URL（引入 StudentListLessonView 类），代码如下。

```
path('lessons-list/<int:course_id>/', StudentListLessonView.as_view(),
     name="lessons_list"),
```

重启 Django 服务，在浏览器的地址栏中输入合适的地址（course_id 在数据库中真实存在），例如笔者测试的 URL 是 http://localhost:8000/course/lessons-list/6/，即可浏览某课程标题内的所有课程内容列表。

另外，读者在创建的 http://localhost:8000/course/course-list/页面上创建浏览课程内容列表的入口应该不成问题，与前面的操作一样，只不过这次修改./templates/course/course_list.html 文件（不再赘述，请读者自行完成）。

至此，完全是复习。下面要在课程内容列表页中增加一个“注册学习”的功能，表示用户

加入该课程的学习。再次编辑./templates/course/slist_lessons.html 文件，重点是增加一些 JavaScript 代码。下面是这个文件的全部代码。

```
{% extends "base.html" %}
{% load staticfiles %}
{% block title %}Lessons List{% endblock %}
{% block content %}
<div class="container">
  <div class='text-center'>
    <h1>{{ course.title }}</h1>
    <p>课程内容列表</p>
    <form action="." method="post" class="form-horizontal">{% csrf_token %}
      <input type="button" onclick="start_learn({{course.id}})" value="Try my best to
learn the course">
    </form>
  </div>
  <table class="table table-hover" style="margin-top:10px">
    <tr>
      <td>序号</td>
      <td>内容标题</td>
      <td>概要</td>
      <td>发布日期</td>
    </tr>
    {% for lesson in course.lesson.all %}
    <tr id={{ forloop.counter }}>
      <td>{{ forloop.counter }}</td>
      <td><a  href="{%  url  'course:detail_lesson'  lesson.id  %}">{{  lesson.title  }}
</a></td>
      <td>{{ lesson.description | slice:'70'|linebreaks }}</td>
      <td>{{ lesson.created|date:"Y-m-d" }}</td>
    </tr>
    {% empty %}
    <p>还没有发布具体内容，请等待。</p>
    {% endfor %}
  </table>
</div>
<script src='{% static "js/jquery.js" %}'></script>
<script src='{% static "js/csrf.js" %}'></script>
<script>
    function start_learn(course_id){
        $.ajaxSetup({
            beforeSend: function(xhr, settings) {
               function getCookie(name) {
                   var cookieValue = null;
                   if (document.cookie && document.cookie != '') {
                       var cookies = document.cookie.split(';');
                       for (var i = 0; i < cookies.length; i++) {
                           var cookie = jQuery.trim(cookies[i]);
                           if (cookie.substring(0, name.length + 1) == (name + '=')) {
                               cookieValue                                          =
decodeURIComponent(cookie.substring(name.length + 1));
```

```
                        break;
                    }
                }
            }
            return cookieValue;
        }
        if (!(/^http:.*/.test(settings.url) || /^https:.*/.test(settings.url))) {
            xhr.setRequestHeader("X-CSRFToken", getCookie('csrftoken'));
        }
    }
  });
  $.ajax({
    type: "POST",
    url: "{% url 'course:lessons_list' course.id %}",
    data: {'course_id':course_id},
    success:function(e){
      if(e=="ok"){
        alert("You have enroll this course.")
      }else{
         alert("Sorry, There are some mistakes.")
      }
    },
  });
 }
</script>
{% endblock %}
```

因为以 Ajax 方式向视图端提交了注册本课程的信息，所以在./course/views.py 文件的 StudentListLessonView 类中要写一个 POST 方法，代码如下。

```
def post(self, request, *args, **kwargs):
    course = Course.objects.get(id=kwargs['course_id'])
    course.student.add(self.request.user)
    return HttpResponse("ok")
```

如此就把用户注册本课程的信息记录到了数据库表中。确保 Django 服务已运行，然后进行调试，比如在 http://localhost:8000/course/lessons-list/6/页面中单击如图 7-17 所示的按钮。

图 7-17　注册学习

随后查看数据库表 course_course_student，就能发现增加了当前用户 id 和课程 id，如下所示。

```
sqlite> select * from course_course_student;
id         course_id   user_id
----------  ----------  ----------
1          6           8
```

上述功能虽然实现了，但还有可以优化的空间，比如判断当前用户对某课程是否已经注册、在页面某个地方显示用户已经注册了的课程。这些功能读者用已经学习过的技能可以解决，此处不再演示，请读者自行学习。

通过课程标题列表，可以查看内容标题列表，接下来就要通过内容标题列表查看具体内容了。其方法同前，只不过继承的类有所不同。这个工作还是留给读者自行完成。

本章主要向读者介绍另外一种基于类编写视图的方法，特别是使用 Django 提供的通用视图类，这种方法能够让我们的代码更简捷。在真实的项目开发中，“基于类”和“基于函数”两种书写视图文件的方法没有优劣之分，可以根据具体的情况选择使用。

7.3.6 知识点

1．模型：null = True 和 blank = True

在数据模型类的属性中，null = True 表示该字段可以为空。对于 CharField 和基于 CharField 的属性（如 TextField 等），如果设置了 null = True，则表示在“没有数据”时，而不是 NULL。只有对于非字符类（比如 DateTimeField），才会保存 NULL。

blank 这个参数负责控制该字段是否允许为空，默认是 blank = False，不能为空；如果设置为 blank = True，则在填写表单时，该输入框可以为空。在进行表单对象检验时，is_valid()函数认同所提交的空字符串。

对于不同类型的字段，要注意这两个应用。比如：

```
models.DateTimeField(blank=True)
```

这样就会报 IntegrityError 错误，必须要按照下面这样写：

```
models.DateTimeField(null=True)
```

下面再列举一个字符串类型的。

```
models.CharField(blank=True)
```

这样做是允许的，并且保存空字符串。

```
models.CharField(null=True)
```

这样写虽然不报错，如前述，但也不会保存 NULL，所以最好别这样用。

更多内容请阅读官方文档：https://docs.djangoproject.com/en/2.1/ref/models/fields/#null。

2．第三方应用

在学习 Python 基础知识时，就讲解了它有丰富的标准库和海量的第三方库（请参阅《跟老齐学 Python：轻松入门》），Django 其实也是 Python 中的一个第三方库。Django 继承了 Python 的优良传统，不仅提供了一些默认的应用（最有名的就是 admin），更有众多的第三方应用等待我们选用，比如 Django Packages 网站（网址是 https://djangopackages.org/），它是第三方应用的

集中营，可以在其中选择适合的应用，用于自己项目，从而提高开发效率。下面列出几个应用，读者或许会用到，其他的应用读者可以自行搜索。

- DjangoUeditor：集成百度 Ueditor HTML 编辑器。
- mptt：树状结构的数据管理。
- django rest framework：REST（Representational State Transfer，表述性状态传递）由 Roy Fielding 博士提出，是一种软件设计风格，不是标准，读者可以查阅有关资料了解详细信息。RESTful Web 服务（也称为 RESTful Web API）是一个使用 HTTP 并遵循 REST 原则的 Web 服务。在 Django 开发的网站项目中，有时候会创建接口、查看或者操作某些数据。django rest framework 就是针对这方面需求的应用。

当然，每个应用是否适合读者的项目，只有自己知道，有时候自己也不清楚，那就只有尝试了，幸好尝试的成本比较低。

3．文档导读

（1）*Django REST framework*，http://www.django-rest-framework.org。

（2）*Deploying Django applications to a production server*，https://www.ibm.com/developerworks/library/os-django/index.html。

（3）*How to deploy with WSGI*，https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi。

（4）*Django FastCGI*，https://www.nginx.com/resources/wiki/start/topics/examples/djangofastcgi。

第8章

结束和开始

恭喜你，看到这里，终于要结束这本书枯燥无味的学习了。

祝福你，看到这里，即将开始精彩无比、苦难重重的码农之路了。

一本书终究是有限的，况且Web开发所涉及的领域非常多，绝非一个实践项目能够涵盖的。因此，笔者在本书中特别列出了“文档导读”，试图作为本书正文的补充，尽管如此，也是“挂一漏万”，幸亏还有互联网，它是一个丰富的资料库，在开发过程中如果有需要，且本书没有涉及（没有涉及的居多）或者读者没有学过，可以在网上查询。

就 Django 而言，本书充其量只是入门，没有涉及的内容还有很多，比如生产环境的部署、接口框架、缓存管理等，本书没有讲解不等于不重要。本书的内容虽然不得不暂告一段落，但是读者的真正开发工作恰恰刚刚开始。

在读者日后的开发道路上，最权威的资料当属 Django 官方网站（https://www.djangoproject.com/）上的文档资料。在阅读完本书之后就算入门了，入门之后就要多到这个网站上看官方文档资料，这是入门之后的修行。如果没有入门，是否可以直接看官网的文档呢？可以，但一般适用于有经验的开发者，对于零基础者就不太适合了。

结束啰唆，祝读者在开发的道路不断积累经验，越走越快活。

汪国真的《热爱生命》，恭录于此。

我不去想，
是否能够成功，
既然选择了远方，
便只顾风雨兼程。

我不去想，
能否赢得爱情，
既然钟情于玫瑰，
就勇敢地吐露真诚。

我不去想，
身后会不会袭来寒风冷雨，
既然目标是地平线，
留给世界的只能是背影。

我不去想，
未来是平坦还是泥泞，
只要热爱生命，
一切，都在意料之中。

如图 8-1 所示，“Django makes it easier to build better Web apps more quickly and with less code”，使用 Django 就是“热爱生命”。

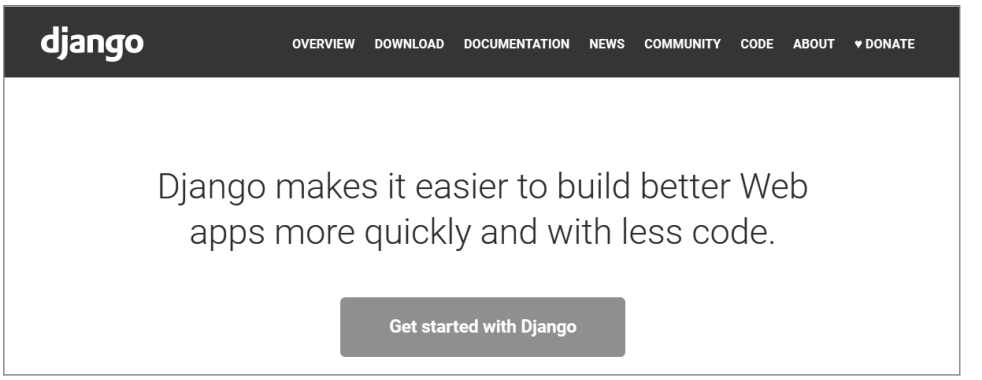

图 8-1　使用 Django 就是“热爱生命”

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E－mail：　dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱　电子工业出版社总编办公室

邮　　编：100036